INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Microservices-Based Monitoring and Decision System for Airport Safety

João Gabriel BUTTOW ALBUQUERQUE
Fatine AZZABI

5 ISS

20/01/2026

Table of contents

Introduction

For this project, we chose to work on an application related to the aviation domain, as both members of the team share a strong interest in this field. In order to identify a realistic and relevant use case, we searched for aerial images of airports and carried out a brainstorming phase to decide on the application to be developed during this course.

By analyzing a top-down view of an airport (as shown in Figure 1), we observed a typical spatial organization of critical infrastructures. In the lower part of the image, the fuel station used to supply aircraft on the ground is generally located at the edge of the airport, allowing easy access for fuel trucks arriving from outside the airport perimeter. In contrast, the fire station is often positioned closer to the runways, in order to minimize response time during critical phases such as takeoff and landing, when the risk of incidents is highest.

Figure 1 - Airport from above

This configuration highlights an important safety challenge: in the event of an incident at the fuel station, firefighters must be alerted and deployed as quickly as possible. Any delay in communication or reaction could allow a dangerous situation - such as a gas leak or fire - to escalate, increasing risks for personnel, infrastructure, and aircraft. Consequently, reducing the response time between the fuel station and the fire station becomes a key objective.

At the early stage of the project, we produced a conceptual sketch to support our brainstorming and better visualize the global system architecture. This sketch helped us clearly identify the different components involved, their physical locations, and the interactions between sensors, decision logic, and actuators. In particular, it highlights the separation between the fuel station area, where sensors are deployed, and the fire station, where emergency actuators are located. This initial sketch played a key role in defining the system architecture and guiding the implementation choices. The conceptual sketch used during this brainstorming phase is presented in Figure 2.

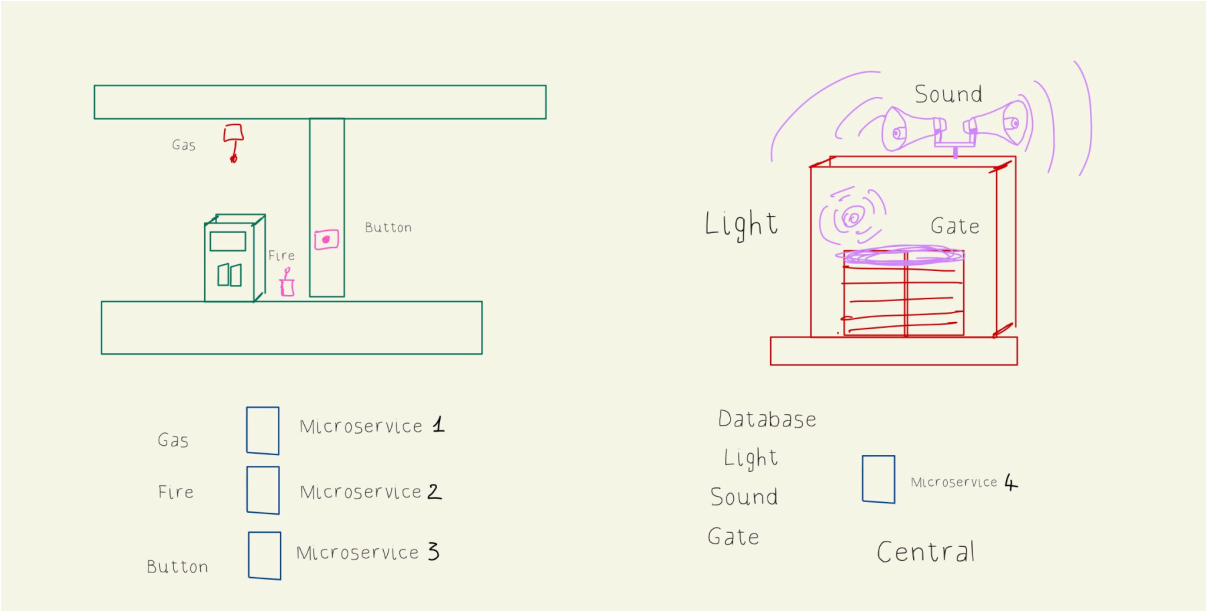

Figure 2 - Project concept sketch

Based on these observations, we designed a system whose objective is to automatically detect abnormal or dangerous situations and trigger an immediate emergency response. Our solution relies on three independent sensors, each implemented as a separate microservice:

- a **gas sensor** to monitor gas levels,
- a **fire sensor** to detect the presence of fire,
- a **manual alarm button** that can be activated by a human operator.

Each sensor runs in its own microservice and exposes its state through REST interfaces. These sensor microservices communicate with a central decision microservice, responsible for evaluating the global situation. When a critical condition is detected, this decision service triggers another microservice controlling the actuators, which simultaneously activates the emergency siren, turns on the warning lights in the fire station, and opens the gate to allow fire trucks to exit immediately.

This microservices-based architecture ensures modularity, clear separation of responsibilities, and fast reaction times, making it well suited for safety-critical applications such as emergency management in airport environments.

Project Creation

The development of this project began with the creation of a structured and modular environment suitable for a microservices-based architecture. In order to reflect the principles of service-oriented systems, the project was organized as a set of independent Spring Boot applications, each one corresponding to a specific microservice. Maven was chosen as the build and dependency management tool, as it provides a reliable and standardized way to configure, compile, and package each service while ensuring consistency across the entire system. The initial setup of the project followed the guidelines and step-by-step tutorial provided by the course instructor, ensuring that all microservices were created with a consistent Spring Boot and Maven configuration.

Each microservice was generated using Spring Initializr with a Maven/Java configuration and the required dependencies (as we can see for example in Figure 1 the creation of decision-engine). This approach mirrors real-world microservice deployments, where services evolve independently and can be developed, tested, and deployed without affecting the others. Using Maven also made it possible to manage common dependencies such as Spring Web, and persistence libraries in a clean and reproducible manner.

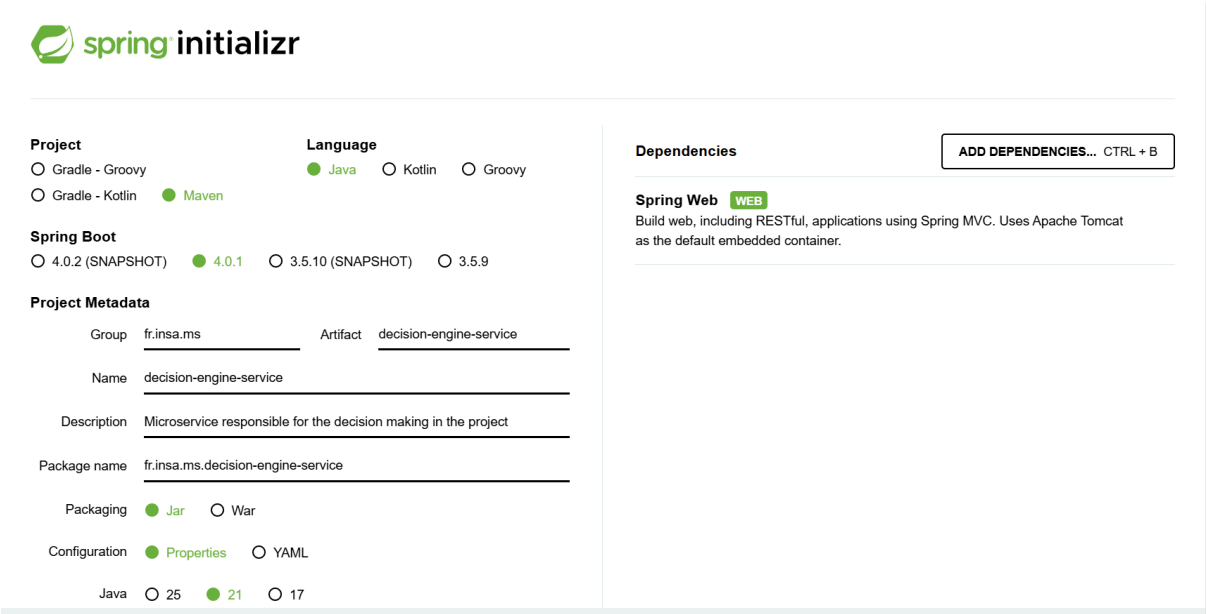

Figure 3 - Creation of the Decision Engine Service using Spring Initializr.

From the beginning, the objective was to ensure that no microservice would depend on the internal implementation of another. Communication between services was therefore designed to occur exclusively through REST interfaces. This constraint forced us to clearly define the responsibilities of each service and to expose only what was strictly necessary through HTTP endpoints. As a result, the architecture naturally evolved toward a clean separation of concerns: sensor services manage only sensor state, the decision engine concentrates all decision logic, the actuator service handles emergency outputs, and the history service is solely responsible for data persistence.

The creation of the project also involved configuring each service to run on a distinct port, enabling all applications to be executed simultaneously on the same machine. This configuration was essential for integration testing, as it allowed the complete distributed system to be executed locally, reproducing a realistic microservice environment. Each service could be started independently, and failures in one service did not prevent the others from running, which is a fundamental property of microservice architectures.

At this stage, the project already reflected the main architectural objectives of the course: decentralization, modularity, and explicit communication between services. By structuring the system as a collection of autonomous Maven-based Spring Boot applications, we established a solid foundation for the subsequent implementation of sensors, decision logic, actuators, and data persistence.

Microservices Description

The architecture of the system is based on a clear separation of responsibilities, implemented through a set of autonomous microservices. Each service models a concrete element of the real-world scenario and exposes a minimal REST interface, allowing the whole system to behave as a distributed and reactive environment. The services can be grouped into four main categories: **sensor** services, the **decision engine**, the **actuator** service, and the **history** service, as shown in Figure 4.

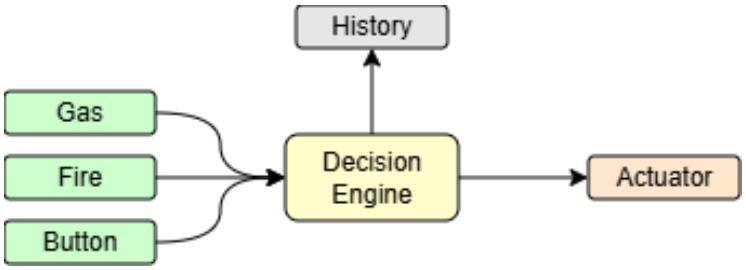

Figure 4 - System architecture

Three independent microservices represent the sensors deployed in the fuel station area. The *Gas Sensor Service* is responsible for maintaining the current gas concentration level. It exposes endpoints allowing this value to be updated and retrieved, simulating the behavior of a real gas detector. The *Fire Sensor Service* models the presence of fire, using a boolean state that indicates whether flames have been detected or not. Similarly, the *Alarm Button Service* represents a manual emergency trigger that can be activated by a human operator. Each of these services encapsulates its own internal state and does not rely on any shared memory or database. Their only role is to provide a reliable and isolated interface through which the rest of the system can query or modify sensor values.

The core of the system is the *Decision Engine Service*. This microservice embodies all the decision logic of the application. It does not store sensor values itself; instead, it retrieves the current state of each sensor by calling the corresponding microservices through REST requests. Based on these values, it evaluates a set of predefined rules that determine whether an emergency situation exists. If one or more conditions are met—such as a high gas level, a detected fire, or a pressed alarm button—the decision engine coordinates the global response. This centralization of logic ensures that all emergency behavior is consistent and traceable, while still preserving the independence of each sensor service.

When an emergency is detected, the decision engine communicates with the *Actuator Service*. This microservice represents the fire station infrastructure and controls three distinct elements: the siren, the alarm light, and the gate. Through simple REST endpoints, the decision engine can activate or deactivate the siren, switch the emergency lights on or off, and open or close the gate. The actuator service does not contain any business logic; it merely executes the commands it receives, mirroring the behavior of physical devices in a real deployment. This design reflects the principle that actuators should remain passive components driven by higher-level control logic.

Finally, the *History Service* is responsible for data persistence. Every significant event in the system - sensor updates, evaluated states, and triggered actions - can be sent to this service, which stores them in a database. By isolating persistence in its own microservice, the architecture ensures that data management concerns do not interfere with real-time decision making. The history service provides endpoints to save new records and to retrieve past events, making it possible to audit the system's behavior and analyze emergency scenarios after execution.

Together, these microservices form a coherent and realistic distributed architecture. Sensors remain simple and autonomous, the decision engine coordinates system-wide behavior, actuators execute physical responses, and the history service guarantees traceability. This separation closely mirrors real industrial systems, where physical devices, control logic, and data storage are clearly decoupled, yet interconnected through well-defined communication channels.

Scenarios

To evaluate the behavior of the system under realistic conditions, five distinct scenarios were defined and executed. Each scenario represents a concrete situation that may occur in the fuel station area of an airport and illustrates how the distributed microservices cooperate. Together, these scenarios demonstrate how sensor updates propagate through the system, how decisions are taken by the decision engine, how actuators react, and how events are recorded by the history service.

Scenario 1 - Normal Operating State

In this initial scenario, all sensors remain in a safe state: the gas level is below the critical threshold, no fire is detected, and the alarm button is not pressed. When the decision engine evaluates the system, it queries each sensor microservice and receives values indicating a normal environment. As a result, no emergency action is triggered. The siren remains off, the alarm light is disabled, and the gate stays closed.

This scenario validates that the system does not produce false alarms and that the decision logic correctly preserves a stable state when no danger is present.

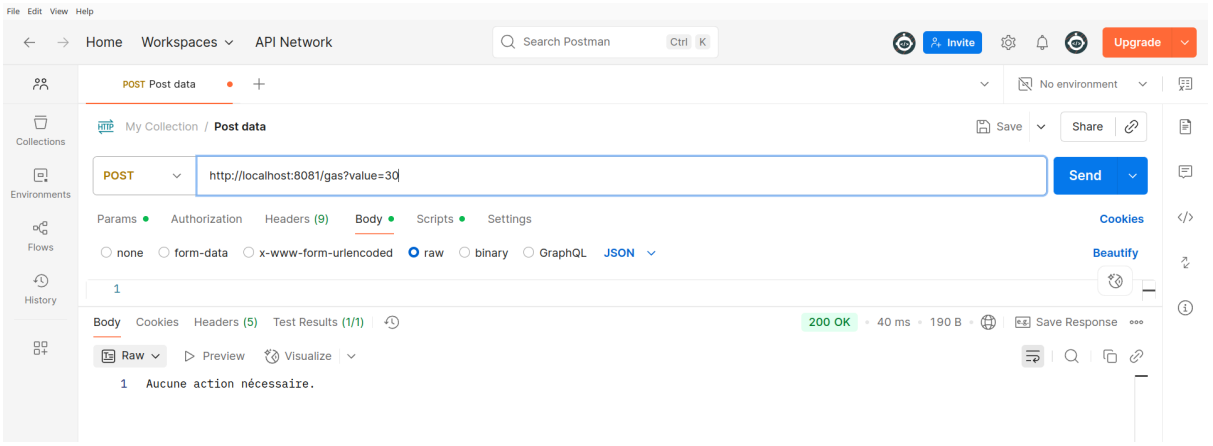

Figure 5 - Scenario 1: normal operating conditions with no emergency detected.

Scenario 2 - Gas Leak Detection

In this scenario, a high gas concentration is sent to the Gas Sensor Service using Postman. This update modifies the internal state of the gas microservice. When the decision engine is invoked, it retrieves this value and detects that it exceeds the predefined safety threshold.

According to the decision rules, this situation represents an emergency. The decision engine immediately contacts the actuator service, activating the siren, turning on the alarm light, and opening the gate. Simultaneously, the event is transmitted to the history service, ensuring that both the sensor value and the triggered actions are persisted.

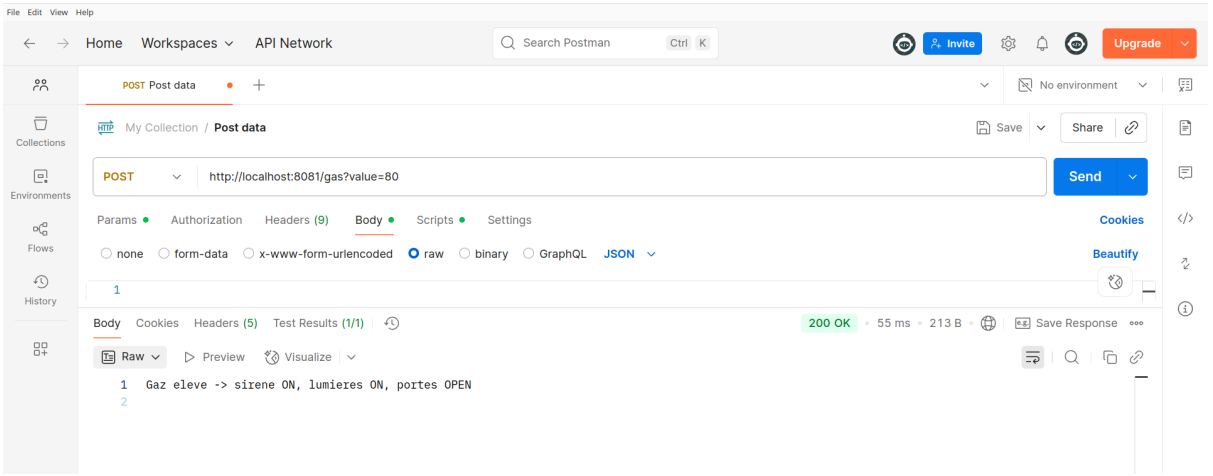

Figure 6 - Scenario 2: gas leak detection triggering siren, light, and gate opening.

Scenario 3 - Fire Detection

This scenario simulates the detection of flames near the fuel station. The Fire Sensor Service is updated with a value indicating the presence of fire. When the decision engine evaluates the system state, it observes that the fire flag is set to true.

This single condition is sufficient to trigger an emergency response, regardless of the other sensor values. The decision engine commands the actuator service to activate the siren, enable the alarm lights, and open the gate. This scenario highlights that multiple independent sensors can initiate the same emergency workflow.

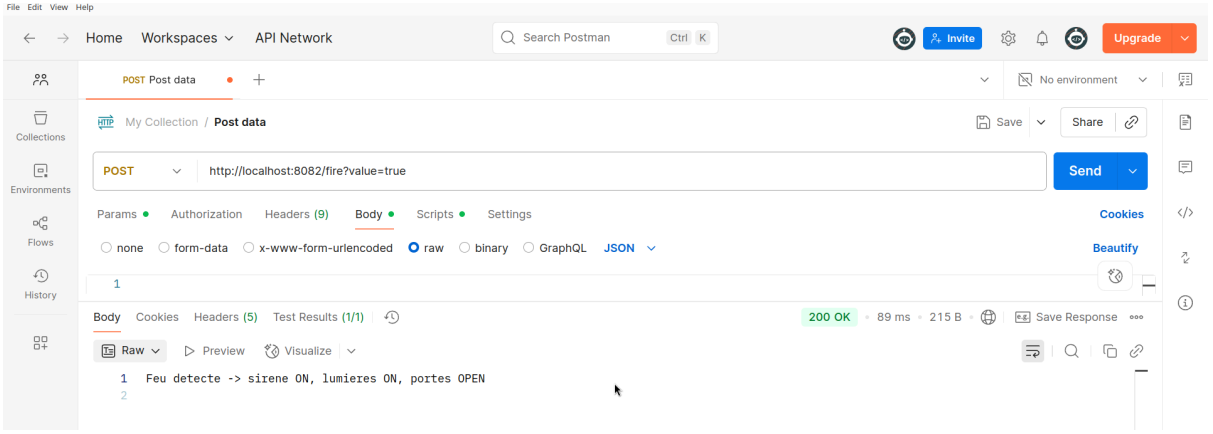

Figure 7 - Scenario 3: fire detection leading to immediate emergency response.

Scenario 4 - Manual Alarm Activation

In this scenario, a human operator presses the alarm button. The Alarm Button Service is updated to reflect this action. Even if no gas leak or fire is detected, the decision engine interprets this input as critical.

During evaluation, the decision engine detects that the alarm button is active and triggers the full emergency procedure. This ensures that human intervention always has priority and can override automated detection mechanisms, reflecting real-world safety requirements.

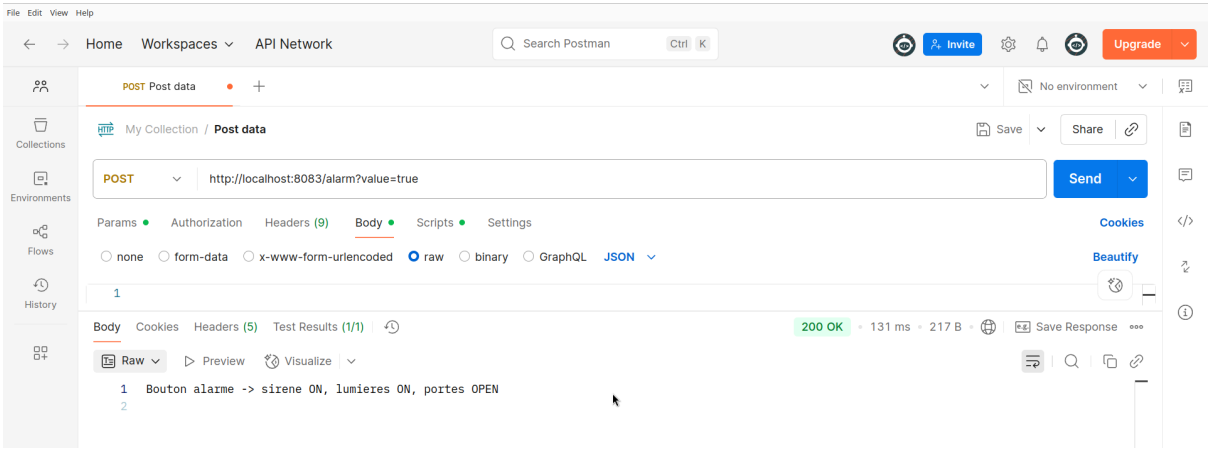

Figure 8 - Scenario 4: manual alarm activation triggering the emergency sequence.

Scenario 5 - Combined Events and Global Traceability

The final scenario combines multiple interactions in sequence, such as first posting a gas value and then activating the alarm button. Each update generates a new evaluation by the decision engine and leads to actuator commands when necessary. For every step, the history service records the sensor readings and the actions taken.

By inspecting the database after this sequence, it becomes possible to observe a complete trace of the system's behavior: multiple readings for each sensor and the corresponding emergency actions. This scenario demonstrates the global coherence of the architecture, where sensing, decision making, actuation, and persistence operate together as a single distributed system.

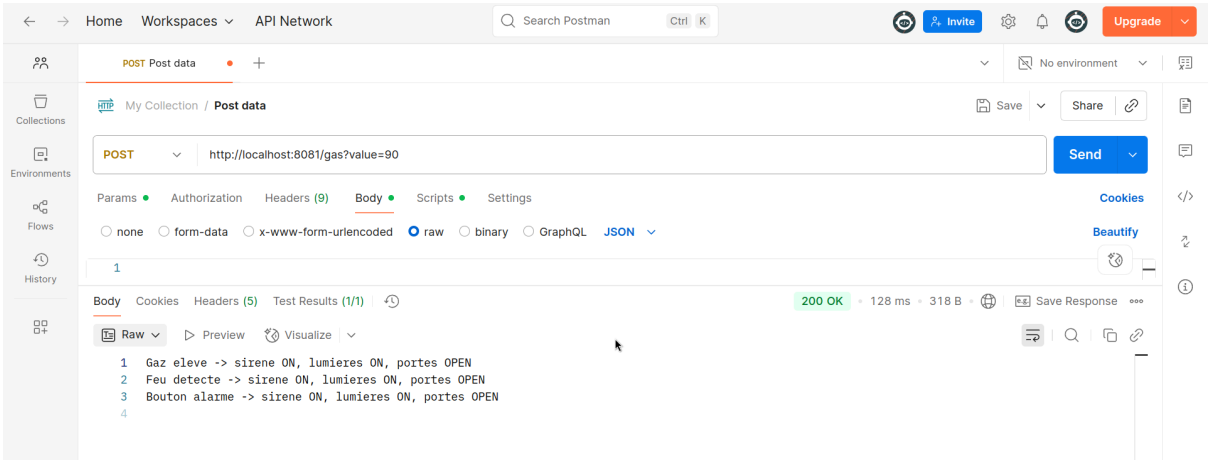

Figure 9 - Scenario 5: combined events and resulting history stored in the database.

Data Storage with MySQL Workbench

In order to ensure traceability and enable post-analysis of the system's behavior, a dedicated microservice was designed to handle data persistence. This service, named the History Service, is responsible for storing every significant event produced by the system, including sensor readings and the actions triggered by the decision engine. The chosen storage solution is a MySQL relational database, managed via MySQL Workbench, as provided by the professor.

Using MySQL as a persistence layer provides a structured and reliable way to keep a permanent record of the system's execution. This is particularly important in the context of emergency management, where being able to reconstruct sequences of events is essential for validation, debugging, and auditing. In our architecture, the database is never accessed directly by the sensor services or by the actuator service. Instead, the History Service acts as the single entry point for persistence, ensuring that storage concerns remain isolated from decision-making and real-time control.

The database model is centered around a single table called *events*, which stores both sensor readings and the corresponding consequences produced by the decision engine. Each record contains an auto-generated identifier, a timestamp, the type of sensor involved, the value measured or reported, and a textual field describing the action that was triggered. This design allows the database to represent both "observation events" (for instance, readings labeled as lecture) and "reaction events" (for instance, emergency commands sent to actuators). The structure of the *events* table, as defined in MySQL Workbench, is illustrated in Figure 10.

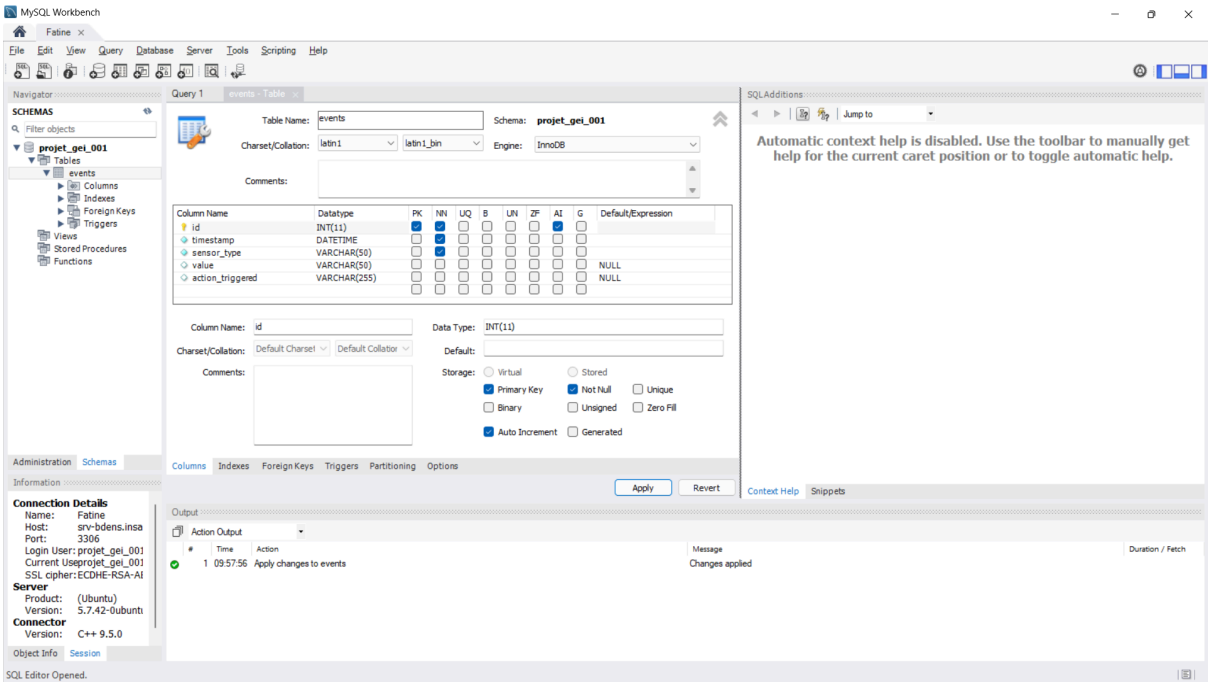

Figure 10 - Structure of the Event table created in MySQL Workbench, showing the fields used to store sensor readings and triggered actions.

Once the History Service is running, the table is created and maintained automatically through JPA entity mapping. Every time the decision engine evaluates the system, it can send relevant information to the History Service, which inserts a new row into the database. This ensures that persistence becomes part of the normal execution flow of the application without introducing additional coupling between microservices. The database therefore represents a complete trace of what the system observed and how it reacted.

MySQL Workbench was used not only to manage the schema but also to validate the correctness of stored data. By inspecting the *events* table after running scenarios, we can confirm that sensor values were collected as expected, that emergency actions were triggered when conditions were met, and that all events were stored with consistent timestamps and meaningful descriptions. This makes the persistence layer a key component of the system, transforming runtime behavior into verifiable evidence that the architecture functions correctly. The result of this final integration test is illustrated in Figure 11, which shows the History Service receiving events through Postman and the corresponding records stored in the MySQL database.

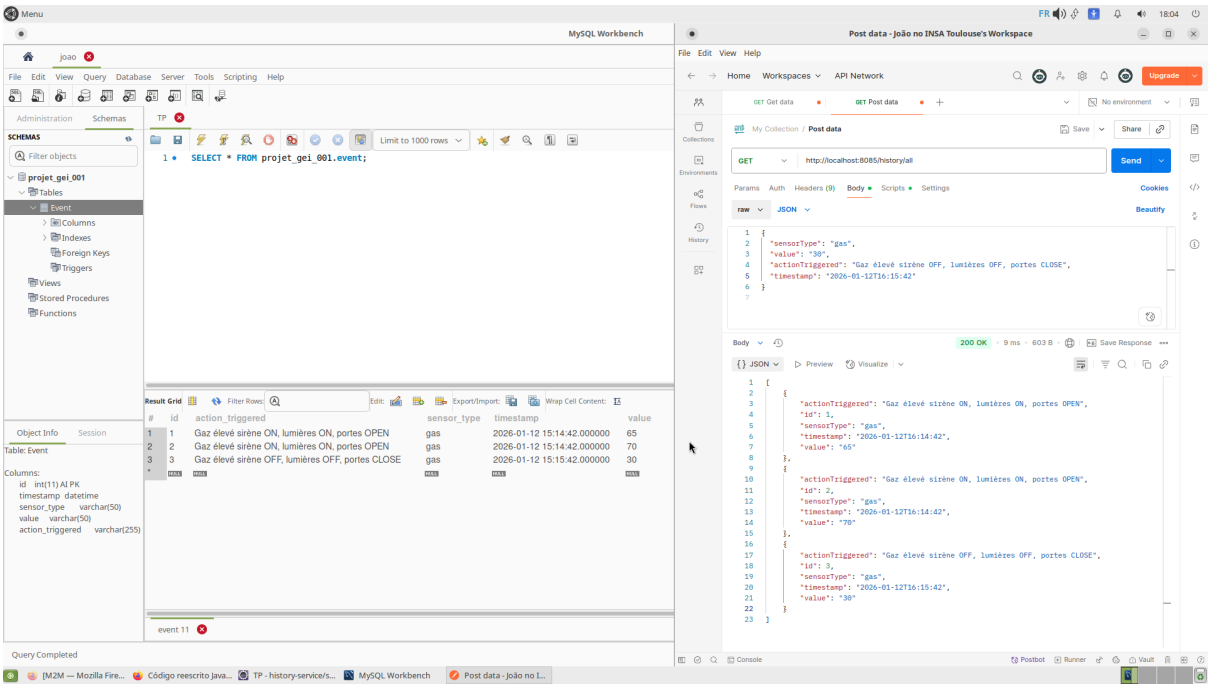

Figure 11 - Final integration test: Postman requests sent to the History Service (right) and corresponding events stored in the Event table in MySQL Workbench (left).

Final Integration and Global Validation

The final phase of the project consisted in integrating all microservices and validating the system as a whole. In order to reproduce a realistic distributed environment, each service was executed independently in Eclipse JEE. The services were started in a precise order to ensure correct dependencies and availability: first the three sensor services (Gas-Sensor-Service-Application, Fire-Sensor-Service-Application, and Alarm-Button-Service-Application), followed by the Actuator-Service-Application and the History-Service-Application. Finally, the Decision-Engine-Service-Application was launched, as it constitutes the central component responsible for coordinating all other services.

This execution order reflects the architectural hierarchy of the system. Sensors and infrastructure services must be available before the decision engine can operate, since it depends on them to retrieve data, trigger actions, and store events. Once all services were running, the system formed a complete distributed environment in which every component could communicate through REST interfaces.

The global behavior of the architecture was then tested using Postman. During this final test, sensor values were published manually to simulate real-world events. First, a safe gas value of 30 was sent to the Gas Sensor Service. This update did not trigger any emergency, confirming that the system remains stable under normal conditions. Subsequently, the alarm button was activated by posting a value of true to the Alarm Button Service. This change of state caused the Decision Engine to be invoked through the /decision/evaluate endpoint.

Upon evaluation, the Decision Engine retrieved the current states of all sensors and detected that the alarm button was active. According to the predefined rules, this situation represents an emergency. The Decision Engine therefore contacted the Actuator Service, which activated the siren, turned on the alarm lights, and opened the gate at the fire station. At the same time, the Decision Engine forwarded all relevant information to the History Service, which stored both the sensor readings and the triggered actions in the MySQL database.

The result of this final integration test is illustrated in Figure 12. On the right side, Postman shows the last request sent to the Alarm Button Service, indicating that the button was pressed. On the left side, MySQL Workbench displays the content of the Event table. Because two sensor updates were sent during this test, the database contains six “lecture” records, corresponding to two evaluations of each of the three sensors. In addition, a final record represents the emergency action triggered by the system, including the activation of the siren, the alarm light, and the gate.

This experiment demonstrates that the entire architecture operates correctly as a coherent distributed system. Sensor updates propagate through the microservices, decisions are taken centrally, actuators respond appropriately, and every step is persistently recorded. The successful execution of this final test confirms that the microservices-based design fulfills its objectives and that the system behaves as expected in a realistic emergency scenario.

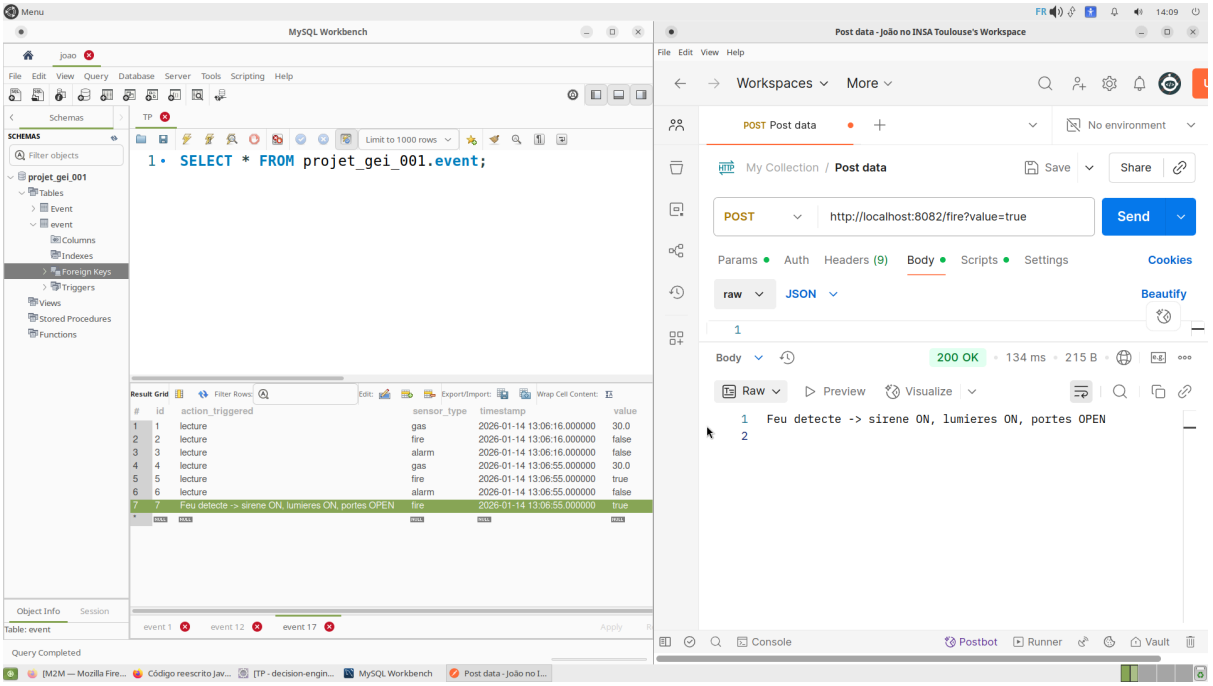

Figure 12 - Final system validation: sensor update sent via Postman (right) and resulting records stored in the MySQL event table (left).

Our project was developed on INSA's computers, and we also uploaded it to a GitHub repository so it can be accessed by everyone via the following link: https://github.com/Azefalo/Architecture-de-service