Automatic Management of INSA's Rooms

Isabella Mi Hyun Kim, Noël Taillard and Poonkuzhali Pajanissamy
5th year ISS - group A2

Outline

Introduction

1 System Architecture
2 Use cases
3 Implementation of the architecture
4 Application program interface
- List of all http requests implemented on RoomMS and LightMS (same as other MS)
- Example of the execution of an action on the lights of a room
- Detail on the microservices created
5 User Interface
6 Project Management

Conclusion

Introduction

The aim of this project was to create an application capable of managing automatically INSA's rooms. The application is capable of automatically turning on/off all the computers in a room, the heaters and closing/opening the shutters. These actions are based on the information received from sensors located in these rooms.

The application is based on a **Service Oriented Architecture** (SOA), which consists in the creation of services (i.e., functionalities) that can be implemented by application components using a communication protocol via a network. These services can be accessed remotely and acted upon.

To create our application, we defined actuators, sensors and services as well as use cases scenarios. The application retrieves data from sensors (time, input from users, etc), and based on these collected information, actions can be triggered. All of these devices are managed by a server and the data is stored in a OM2M platform. The OM2M platform is an abstraction of our actuators and sensors as well as INSA's rooms.

In this report, we'll also explain some project management methods we used for our organisation, the Agile Scrum methodology and the Icescrum organization tool. The source code of the whole project is available on a GitHub repository, along with a READ.me describing **how to set up** our system, and more, to pursue this report:

https://github.com/noeltaillardat/RoomsManagement

1 Use cases

While we could have the opportunity to build a *smart building*, with autonomous behaviour based on predefined scenarios, we decided to provide something more: the ability for the user to control the building and create his or her *own* scenarios. Regarding what sensors and actuators could be controlled, we decided our Proof-of-Concept project to aim at some realistic and useful features. Here are some devices that can be automated, and that we which to be able to provide *as a service*:

- the lights, because managing smartly the lights of the building could prevent someone to manually go through each room switching off the lights at night (we also noticed that often, lights remain switched on in GEI at night, because some students forgot to turn it off after leaving)
- the computers, because here again, switching off all computers at night could save energy, and switching them on before the beginning of practical works would be a way to save time, since it usually takes some to turn on the computers, and it is a time lost during the practical works
- the heaters, because students and teachers like to work in warm, comfortable environments
- the shutters of the windows (here after shortened as 'windows'), since automated closing windows are actually complicated and useless (people do not forgot to close windows), however electrical shutters do exist, and would be able to replace fastidious opening and closing of hundreds of INSA's windows each day all over the campus

There, our aim is thus to make those actuators controllable manually, based on user-defined schedules, and base on temperature. However, developing our project with an Agile method (see last part), if we manage to develop our application, we did not implement the temperature Microservice we intend to deploy, since it was not a priority story, and it base been postponed for a potential, future sprint.

2 System Architecture

Regarding our architecture, we first have to present the physical architecture of the connected devices—or here, how we virtualize them. We indeed used **OM2M**, an IoT middleware, to handle the physical devices (which in the project remain virtual, but representing as if real with OM2M database).

We defined three different layers to deploy our architecture the first one being INSA's campus, for which we defined an Infrastructure Node (IN). From the top IN, we defined a second layer composed of Middle Nodes (MN), these nodes placed as gateways across INSA's building, one per floor. Here, we deployed three MNs, one per GEI's floor, to cover all devices that could be inside the building. On a third layer, we defined several Application Entities (AEs) connected to the network through the MNs. Each AE represent a room that can be monitored with our application. It's in this layer that our sensors and actuators are located, their stated being stored in OM2M database as Content Instances of Containers, one per device.

From their, sensors and actuators communicate with the gateway (MN) through a wireless connection. The information collected is then transmitted to our IN (could be in the cloud) which provides its API to our microservices (MS). Finally we created a graphical user interface

(UI) that allows the user to see information about the devices as well as manually control the actuators, are schedule their behaviour, as explained before.

In the figure below we show a diagram containing the actual deployment of the architecture of our system:

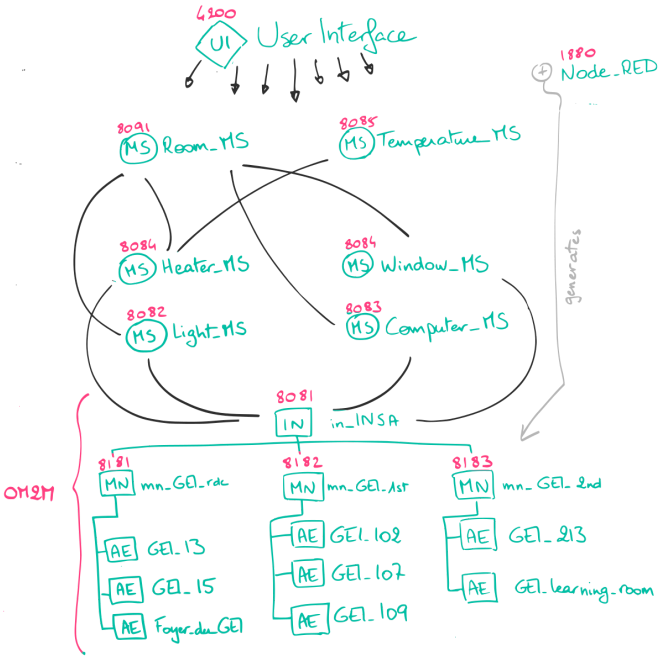

Figure 1: Architecture of the Service Oriented application

3 Implementation of the architecture

We describe here a short overview of the implementation of the architecture. To get a full description of how to set up our Service Oriented Architecture project, please consider our detailed READ.me provided on our [GitHub repository](). We used Node-RED to create all our AEs, Containers (CNTs) and Content Instances (CINs) defined layers in OM2M. As shown on Figure 2 below:

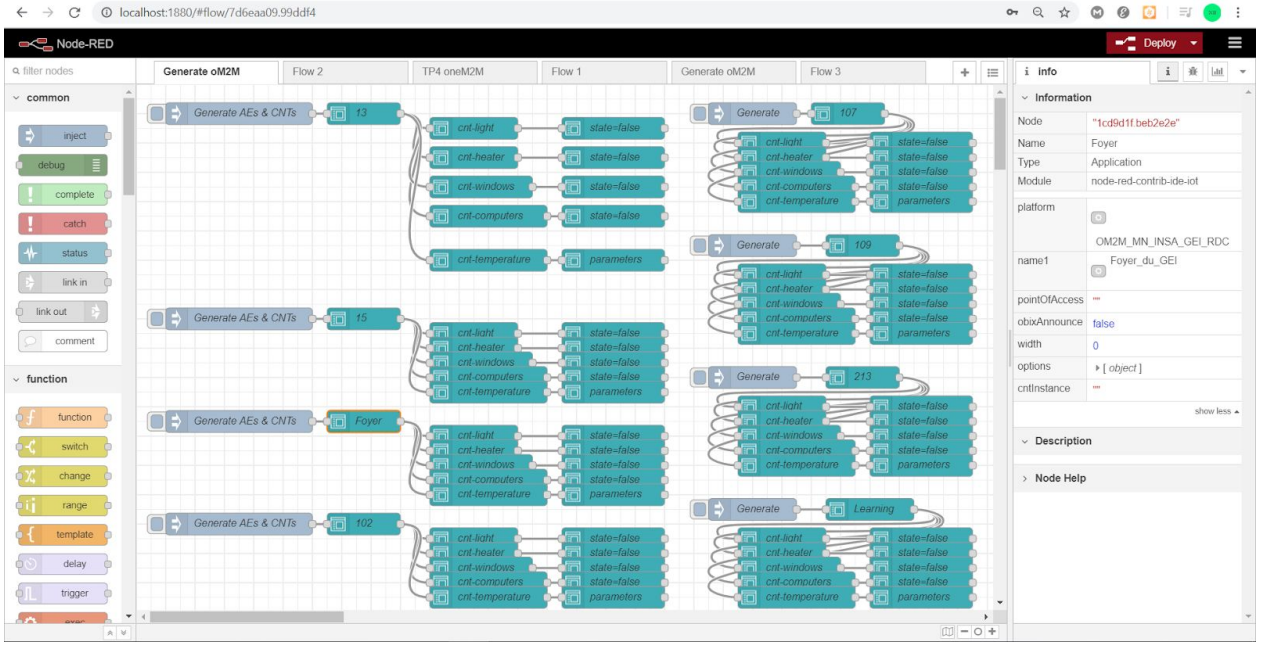

Figure 2: Creation of the AEs and their data

With the architecture deployed in OM2M we could create an abstraction of GEI's rooms, where we could visualize the state of our actuators and implement actions on the lights, heaters, windows and computers of the rooms based on our inputs. In Figure 4 below we show the OM2M platform with the rooms where the actions are executed.

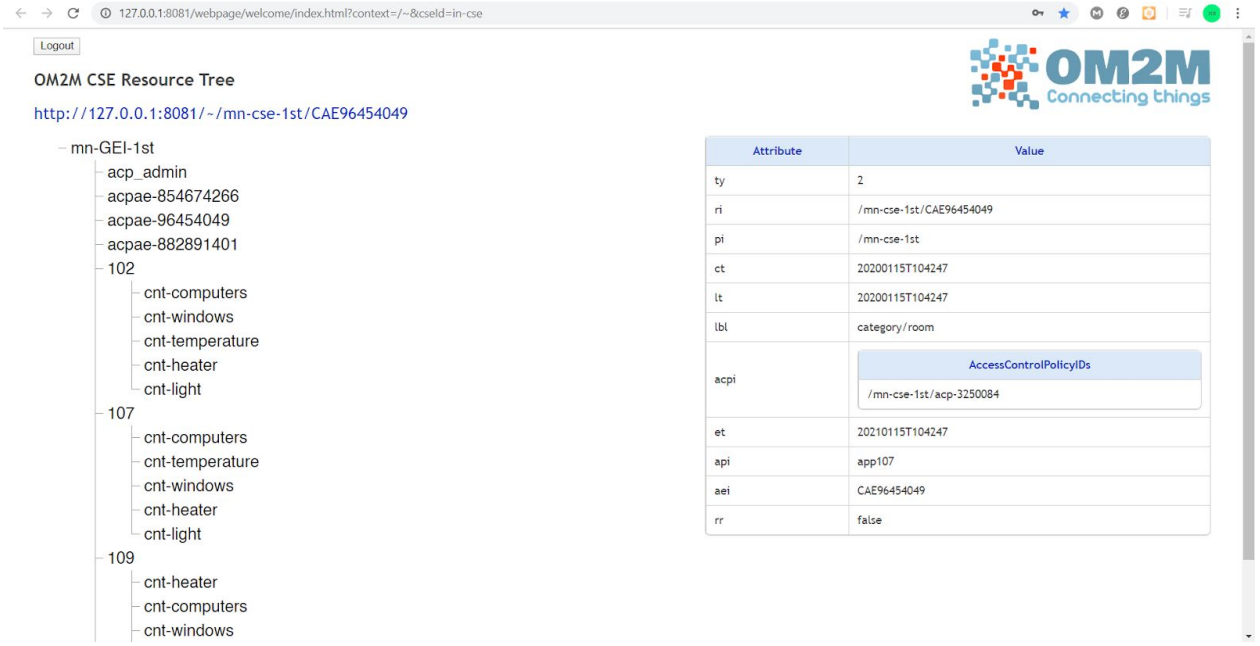

Figure 3: OM2M webpage

We used Eclipse Java EE IDE Neon 3 to develop our **Rest API** and the **Spring boot** framework to ease MS creations and deployments. We created microservices for each of our actuators, so we have: lightsMS, heatersMS, windowsMS and computersMS. These microservices manage the POST, GET, PUT and DELETE http requests. So, through these microservices we can send http requests to execute actions on application entities.

To facilitate the interaction with the user and to test our deployed system, we created a user interface. The UI has been developed using the framework Angular. It is thus coded in TypeScript, HTML and CSS. Thanks to the possible offers by the framework, we can develop individual components that would be dynamically duplicated depending on what actuators are available, and what rooms are available.

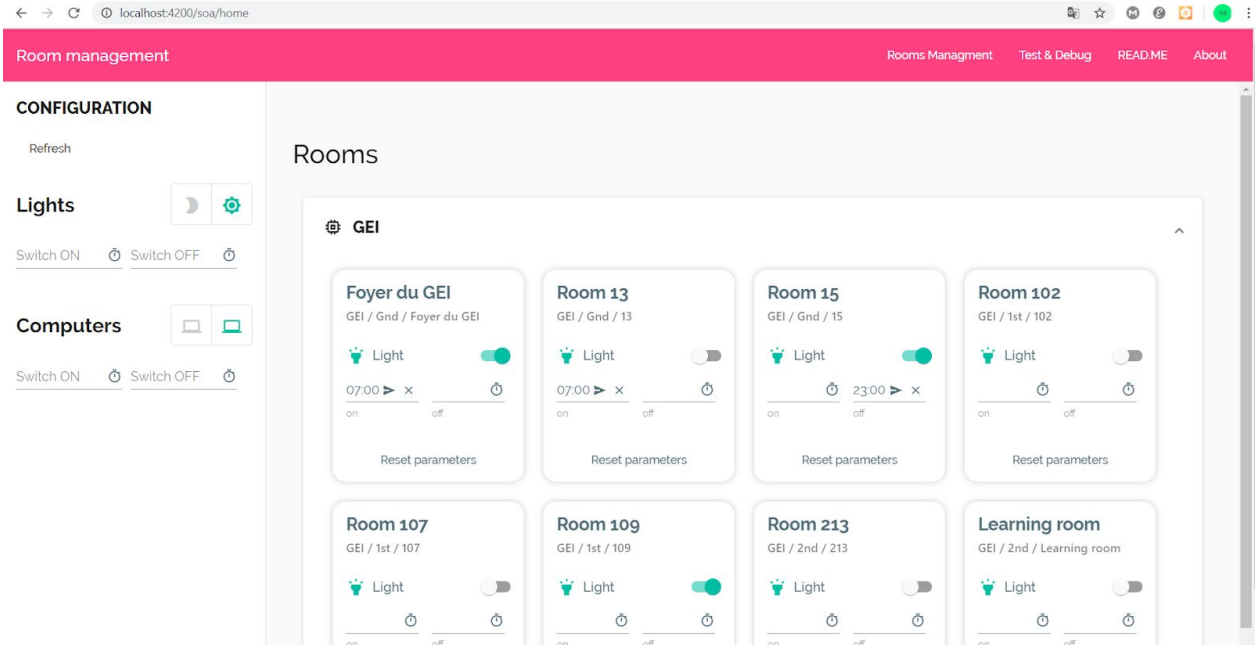

Figure 4: User interface to interact with the system

4 Application Program Interface (API)

We structured an http format to retrieve and send data to our actuators, the path follows a hierarchical order: first we specify “INSA” as a base location, then the name of the building we want to interact with (GEI), after that the room This path gives us our actuators, since we control all the actuators INSA’s room (in the code, we refer at the path *{building}/{room}* as a “roomID”). This is the path for a GET request. For posting a request, we also used the GET request: the content of our requests being simple, it could fit in the url without any need for some other message, and it eases development of tests, MSs, and UI. To build such a request, we just added a state to the path indicating if we want to activate or deactivate the actuator in question.

Because we have to know on which floor is a room to access it (connected to a specific MN) implemented the Rooms Micro Service that could be asked for retrieving floors of rooms, based on their building and name, since the aim is to control rooms easily, just by knowing their name and building. RoomMS acts as a *dictionary* of GEI’s building rooms.

Here’s an example of the structure of a GET request to get the state of the lights of a room:

/INSA/{building}/{room}/light/

And next is an example of a request to turn the lights on/off:

/INSA/{building}/{room}/light/{state}

List of all http requests implemented on RoomMS and LightMS

(as an example for the actuators MS).

This list is present as another tab (next to the RoomManagement, main tab) in our User Interface, and used to check and debug the behaviour of the UI and MSs:

Figure 4: List of all implemented http requests, available in a tab of the UI

Examples of the execution of an action on the lights of a room

So, for example, to interact with the lights of a room. Here's what happens:

1- We can use a GET to retrieve the current state of the lights, using the http request path; here, we will take an example with the Foyer_du_GEI, with the light actuator:

localhost:8082/INSA/GEI/Foyer_du_GEI/light/

2- To ask the lamp its state, the MS needs to know its location, so it sends a request to the RoomMS:

localhost:8091/INSA/GEI/Foyer_du_GEI/floor/

3- The RoomMS returns the floor…
4- …that LightMS can use to retrieve the address of the mn-cse of the ground floor, refers here as in French, 'rdc' (to retrieve the last instance, in OM2M, use the *cnt-name-la*, and *cnt-name-ol* for the oldest):

localhost:8181/~/mn-cse-rdc/mn-GEI-rdc/Foyer_du_GEI/cnt-light-la/

4- The state is returned to the MS…
5- …that returns it again, back to the UI (or the browser where the GET http request

The state of the light can be seen in OM2M under the CNTs of the sensors as well as on the GUI, since the GUI as a refresh button calling all gets methods, and will automatically update the displayed states of actuators few times per minute.

Another example, if we want to schedule the light of this room to automatically switch off at 11pm (23:00), when all students must have left the building:

1- The request is sent: *on* or *true*, and *off* or *false* are valid states for the MS, and the time is defined as a simple timestamp in the format “hh:mm:ss”, the seconds are actually not taken into account).

localhost:8082/INSA/GEI/Foyer_du_GEI/off/23:00:00

2- The LightMS store this value (GEI>Foyer_du_GEI>off>23:00:00), and compare all of stored values with the time being in its main thread. While time has come for students to go home, the LightMS reproduce all steps from the second one of the example above.

Detail on the microservices created

In the code shown in the figure below for the lights resource, we show how the microservice receives and sends the HTTP requests to change the content of the OM2M AE. The same was done for the windows, the heaters and the computers.

```
@CrossOrigin
@GetMapping(path="/INSA/{building}/{room}/light/{state}")
public static int setLightStatus(@PathVariable String building, @PathVariable String room, @PathVariable String state) {
    String newState = "false";
    RestTemplate restTemplate = new RestTemplate();
    String floor =  restTemplate.getForObject(
            ROOM_MS + "/INSA/" + building + "/" + room + "/floor",
            String[].class)[0];

    if (floor.equals("404")) {
        return 404;
    } else {
        switch (state) {
            case "true":
            case "on":
                newState = "true";
                break;

            case "false":
            case "off":
                newState = "false";
                break;
        }
        try {
            HTTPManagement.sendPOSTActuator(
                // IN INSA
                IN_INSA +
                // MN corresponding to the floor of the good building
                "/mn-cse-" + floor +
                "/mn-" + building + "-" + floor +
                //
                "/" + room +
                // getLight
                "/cnt-light",

                // state
                newState);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return 200;
    }
}
```

Figure 6: Part of the code of a microservice to change the lights in a room.

Another functionality that we implemented on the microservices was the scheduling. With this the user can provide a scheduled time, which is stored in our system, to execute an action on our actuators. When a schedule is sent by the user, we receive an http request with

the coordinates (building, room) of the actuator as well as a timestamp indicating the time the action shall be executed. The microservice stores this schedule and compares to a real time clock. When there's a match between the timestamp and the clock time, the action is executed, using the method presented above. That part of the microservice's code is shown in the figure below.

```
// Timestamp format: xx:xx:xx
@GetMapping(path="/INSA/{building}/{room}/light/{state}/{timestamp}")
public void setLightStatusOnSchedule(@PathVariable String building, @PathVariable String room, @PathVariable String timestamp, @PathVariable String state) {
    //Light light = lights.get(building + "/" + room);

    //arrays.add(array_test);
    boolean newSchedule = true;

    for (String[] scheduled : arrays) {
        if (scheduled[0].equals(building) && scheduled[1].equals(room) && scheduled[2].equals(state)) {
            scheduled[3] = timestamp;
            newSchedule = false;
            System.out.println("Array already exists!");
        }
    }

    if (newSchedule) {
        // Creates an array with the room, the state and the timestamp
        String[] array_test = {building, room, state, timestamp};
        arrays.add(array_test);
        System.out.println("Schedule added to list");
        System.out.println(array_test[3]);
    }
}
```

Figure 7: Part of the microservice's code to execute actions on actuators based on a user schedule entry

5 User interface

Through this interface, the user is able to visualize all the rooms in the building and the state of their respective actuators. The user can either manually control the status of the actuators of the rooms (turning on/off lights, heaters, computers and opening/closing windows) or by scheduling those actions.

The main panel also offers the possibility to act on all actuators at once, with a configuration panel, highlighted below:

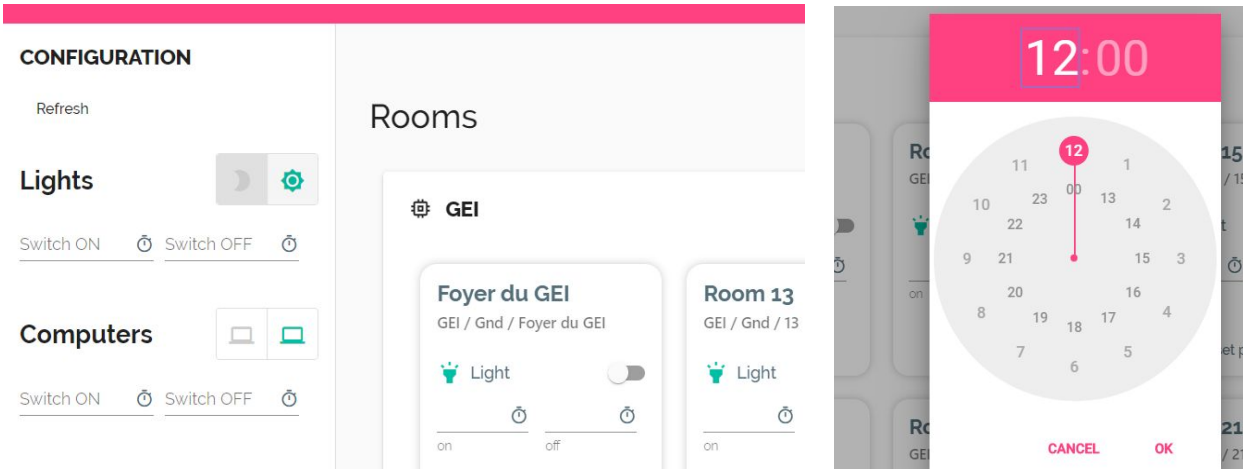

Figure 8: Zoom on some features of the UI

A second panel, as introduced above, present all http requests implemented in our microservices, and an easy way to test them (like a form), to both check the validity of the display of the main panel and check that the microservices do work.

6 Project management

During this project we followed the Agile method, creating sprints using the Icescrum platform. The idea of this method is to allow a more flexible development, establishing the next objectives and tasks based on the current advancement of the project. Also at the end of each cycle ("sprint"), we should be able to have a small functioning part of what is the whole system that we want to achieve in the end.

At the beginning of the development of the project, we dedicated a session to define our main goals to the final project, what we wanted to present and the use cases we found more important to address. At the end of that session, we defined some stories on IceScrum (see figure below).

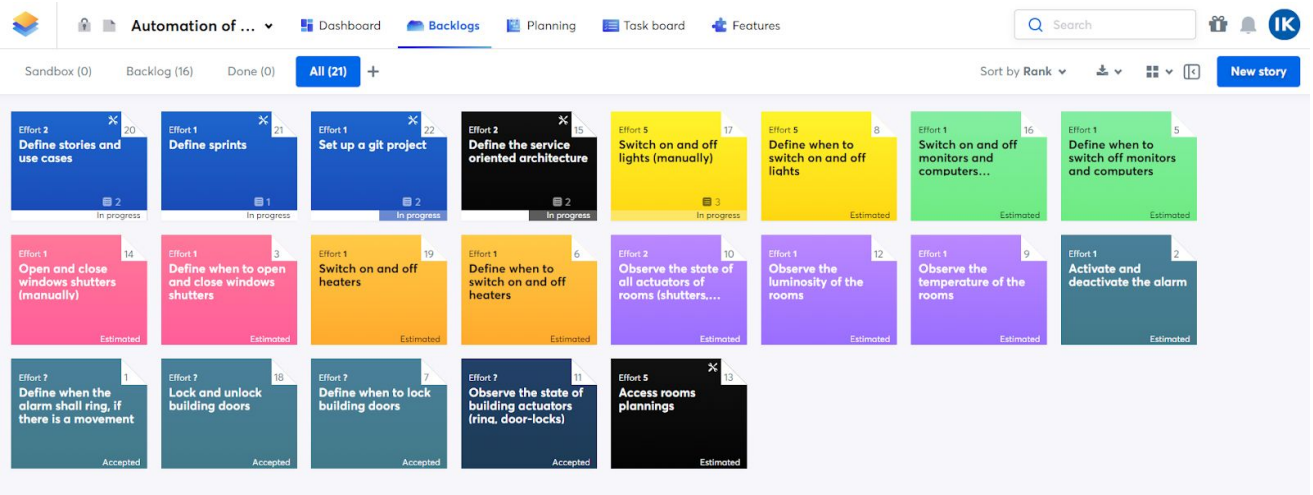

Figure 9: Stories defined in our first sprint using the IceScrum platform.

Due to time constraints, we chose to modify some objectives defined in our first sprint planning shown in Figure 9 to something more realistic to the time we had in our time slots to dedicate to this project. We first organised our project in 3 sprints; because we were far from our objectives, we added the last one to finish to implement, not all stories we defined, but the most valuable ones remaining: a perfected UI, and an autonomous man:

Sprint 1	- Our main goals for this project - Sensors and actuators of the system - Main use cases to address
Sprint 2	- Creation of a first microservice, executing a simple switch functionality - Setting up the OM2M
Sprint 3	- Integrating the microservices to the OM2M - Duplicate the first MS for more services - Creating a small GUI
Sprint 4	- Implementing the scheduled management - Enhancing the UI

Conclusion

At the end of our project, we were able to provide a working application capable of managing INSA's rooms, and some meaningful services interacting from IoT middleware to web applications framework and going through Microservices developed in Java. This project helped us to understand how software oriented architecture works from bottom to top.

It also highlight how planning such an architecture requires time, and how defining priorities, and work with an Agile method like scrums, allow to have—either at the end of sprint 2, 3, or 4—if not the perfect application we dreamt about, at least a working application, getting better by being more valuable sprint after sprint.