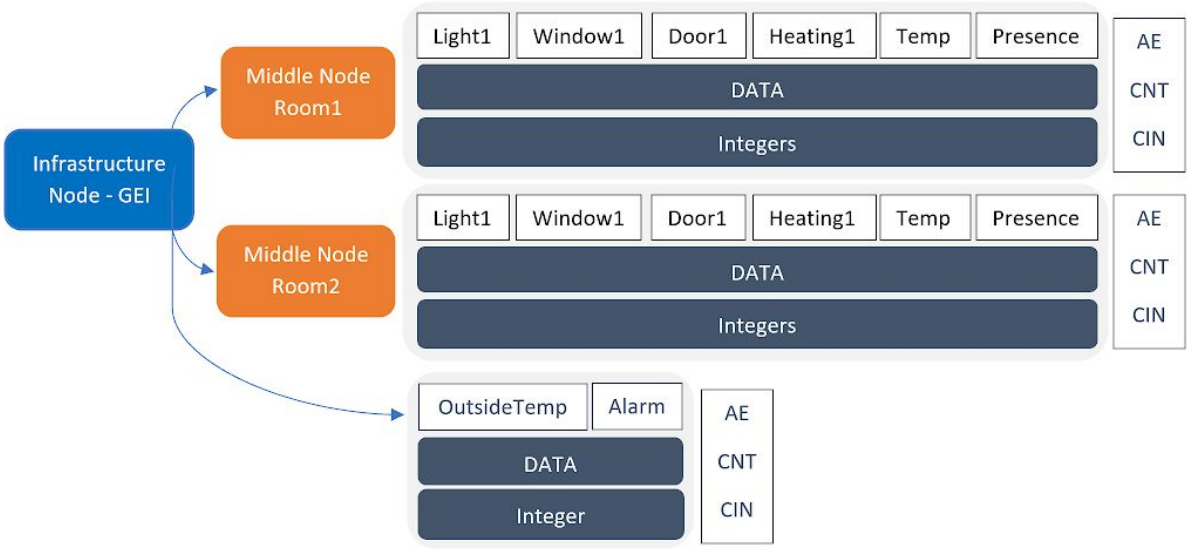
Service Oriented Architecture Report
Automatic management of INSA's rooms

The objective of the project is to develop an application able to manage buildings in a smart way. In order to do that, each room would be equipped with different types of sensors and actuators. The application will implement scenarios which depending on the values retrieved from the sensors will manage the different rooms of the buildings via the actuators.
This project also gave us the opportunity to work using the Agile method through Jira software, improving our teamwork skills. We will present the design of the different architectures and describe the organisation of our project through Jira.

I. Architecture conception

1. OM2M architecture

In our case we decided to apply it to our own department, the GEI, containing two rooms. We simulated arbitrary chosen sensors and actuators through the OM2M platform with which our java application interacted. The base of the architecture is the infrastructure node, here it represents the building, the GEI. It is composed of the rooms which are middle nodes (MN). In each MN we can find the different actuators and sensors which are represented through Application Entities (AE). Each AE contains a container "DATA" (CNT), itself containing Content Instances (CIN) which is the data itself. For simplicity's sake, the information sent are simple integers (set to 1 for OPEN/ON and 0 for CLOSED/OFF).

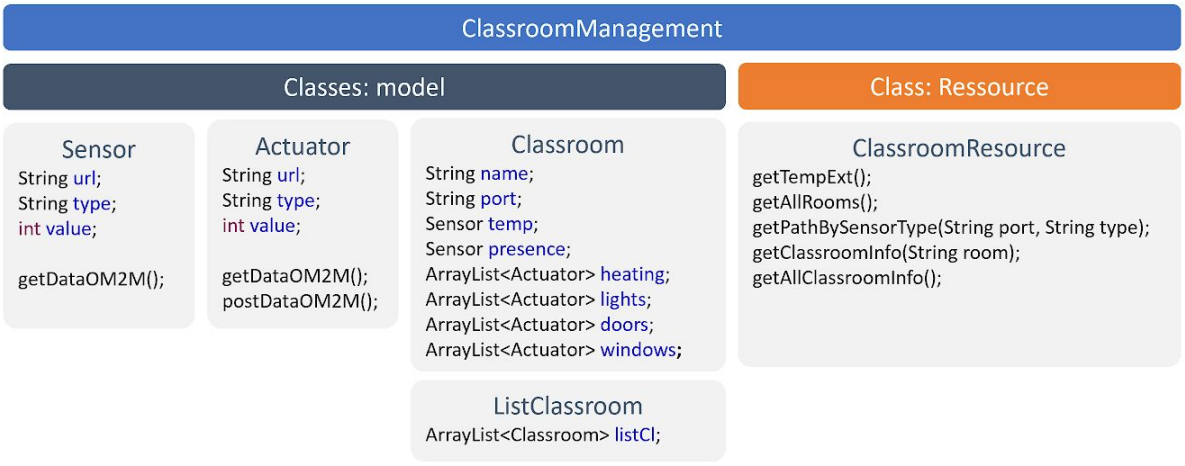


II. Services architecture

The application is divided into three main services associated with the ports 8081, 8082 and 8083: ClassroomManagement, ScenarioManagement and InterfaceManagement.

1. ClassroomManagement

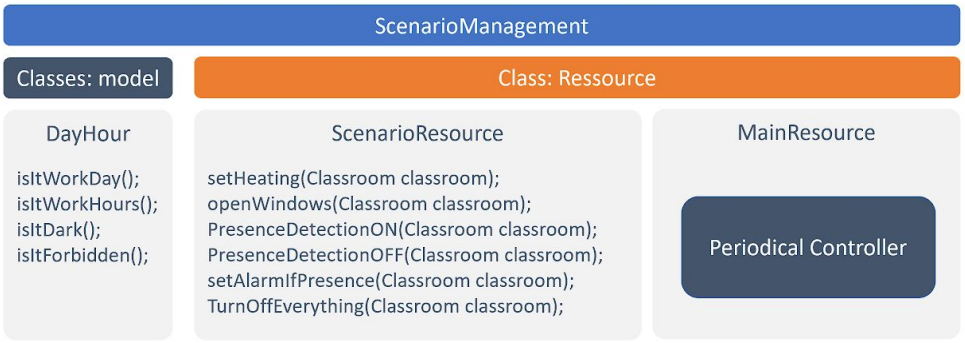
This service manages all the information concerning the actuators and sensors of the rooms. It is composed of four model classes which are used by the main class ClassroomResource. All the functions of the main class are associated with the tag @GetMapping("") in order to access it from a different service.



- public String getTempExt() : returns the outside temperature retrieved on OM2M through a Rest request
- String[][] getAllRooms() : executed starting the service : initializes the global variable *portRoomAssociation* by associating the name of the room to each port (MN ports) given in the *ports* variable.
- public String getPathBySensorType(@PathVariable String port, @PathVariable String type) : returns all the path of the sensors/actuators of a given *type* for a given *port* (room).
- public Classroom getClassroomInfo(@PathVariable String room) : returns a Classroom with all the information of the room (ex : "Room1"). Finds the port number thanks to the given room (cf getAllRooms). Stores in a Classroom variable the information retrieved through a Rest request.
- public ListClassroom getAllClassroomInfo() : returns a ListClassroom containing all the information of all the rooms.

2. ScenarioManagement

This service implements the different scenarios, and runs them periodically. In addition to the model classes seen before, we implement the DayHour class which will help us know when to run a specific scenario. Each scenario takes a Classroom as an argument.



- int setHeating(Classroom classroom) : if the temperature of the room is below 15 then it sets the heating to 1 on OM2M which supposedly turns it ON.
- void openWindows(Classroom classroom) : if the outside temperature is between 18 and 35 then it opens the room's windows.
- void PresenceDetectionON(Classroom classroom) : if there is someone in the room it turns the lights ON.
- PresenceDetectionOFF(Classroom classroom) : if there is no one in the room it turns the lights OFF.
- setAlarmIfPresence(): if there is someone, it sets the alarm.
- TurnOffEverything(Classroom classroom): it sets every actuator in the room to 0 (windows, doors, lights, heatings).

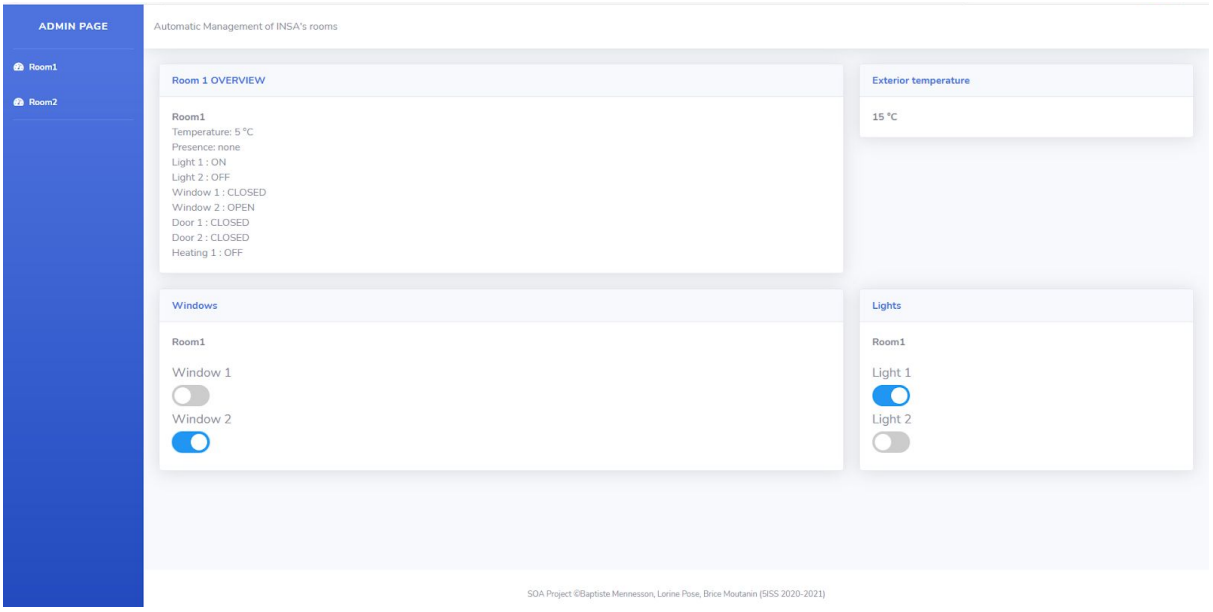
The mainResource retrieves the information of all the classrooms. For each classroom, depending on the day and/or the time of the day, it will call the different scenarios periodically. Depending on the scenarios, they will be called each 10 seconds (PresenceDetectionON for instance) or only each 2 minutes for others (PresenceDetectionOFF for instance).

3. InterfaceManagement

Due to limitations from origin constraints during post requests in javascript, we choose to implement this service as an intermediary between the OM2M platform and the dashboard (implemented in html/CSS/Javascript). Allowing the javascript to continue to work with GET requests whenever an action to change a value occurs, while the associated java service translates these GETs to OM2M POST requests.

III. Dashboard

The Dashboard is a HTML page where the user will be able to see information about the rooms and to interact with actuators.



In the left section of the page, there will be the different rooms available. When the user clicks on a room, the corresponding HTML page will be displayed, and all the information about this room will be loaded, thanks to AJAX Get requests (screenshot on the right).

```
$(document).ready(function(){
  $.ajax({
    crossDomain:true,
    type: "GET",
    url: "http://localhost:8081/GEI/Room1/info",
    success: function (data) {
      var text = `<b>${data.name}</b><br>
                  Temperature: ${data.temp.value} °C<br>
                  Presence: `;
                  if(`${data.presence.value}`==0){
                    text+=`none<br>`;
                  } else {
                    text+=`yes<br>`;
                  }
      //LIGHTS
      for (var i = 0; i < `${data.lights.length}`; i++) {
        text+=`Light `+(i+1)+` : `
        if(`${data.lights[i].value}`==0){
          text+=`OFF<br>`;
        }else{
          text+=`ON<br>`;
        }
      }
```

So for each room, the user can find information like the actual temperature, the presence of someone inside the room represented by a boolean, and the state of each lights, doors and windows of the room. There is also the outside temperature.

```
function actuatorCheck(selectObject) {
  console.log(selectObject.id);
  var data;
  var url="http://localhost:8083/interface/post/"+selectObject.id+"/";
  console.log(url);
  if(selectObject.checked === true){
    $.ajax({
      crossDomain:true,
      type: "GET",
      url: url+"1",
      success: function(msg) {
      }
    });
  } else {
    $.ajax({
      crossDomain:true,
      type: "GET",
      url: url+"0",
      success: function(msg) {
      }
    });
  }
}
```

Then, the user will find on this dashboard different cardboards for the lights and the windows. Each light and window are represented by a toggle, and the toggle's state is synchronized to the real light or window's state, also thanks to previous AJAX Get requests.
If the user clicks on a toggle (see screenshot on the left), it sends a GET request that will be managed by the InterfaceManagement class we have seen previously. Thanks to this management, it will transform the GET request into a POST request, that will modify the value of the object the user clicked on.

Some functionalities have been added to the dashboard to make it more reactive, like an automatic refresh of the information displayed on it.

IV. Agile method through Jira

To help the development of the application to turn out all right and to organize the different tasks we had to do, we chose to follow an Agile Scrum Method thanks to Jira.
This methodology relies on incremental development divided in two to four weeks sprints, during which the goal is to develop the most important features. These features, defined by a client as “user stories”, can be seen as a list of tasks to develop. This method is led by scrum meetings necessary to plan the sprints and associate priorities to the user stories and tasks, in order to chronologically organize them. The advantage of this method and its short sprints is to often present the product, collect feedback and deliver sprint retrospectives.

In order to achieve our application’s development, we firstly defined the user stories associated with the previously listed scenarios and their corresponding tasks. We also attributed priorities to these tasks. A backlog helped us to constantly be aware of our to-do list.

Once we defined the different stories and tasks, we started a first sprint of four weeks. The goal of this first sprint was to set up the tools we were going to use: imagine and implement the OM2M architecture, establish the git of our project, organize the base of the services…
During this first sprint we also coded the tasks associated with the fully automated stories, for example “*As a client, I would like that the heating turns on if the room temperature is below 15°*”.

The realization of this sprint helped us to highlight the most important remaining features and allowed us to organize a second sprint. This second sprint was focused on implementing the dashboard of our project, its associated actions and the displayed information.