Temperature Rest Architecture

Lilian Aguttes
Zacharie Hellouin
5ISS A2

Introduction

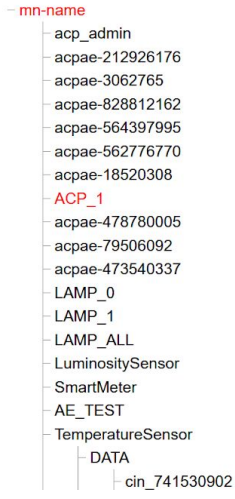
In this report, we are going to explain how we created a Representational State Transfer (REST) architecture to manage the temperature in several rooms. Our goal was to maintain the indoor temperature between 20°C and 25°C.

To achieve this regulation, we decided to use an agile scrum method and divided our project in 3 main sprints. The objective of the first sprint was to implement a REST architecture able to turn on/off the heating inside a room if the temperature is under 20 degree. The second one was to open and close the windows in the room if the temperature is higher than 30 degrees inside. The last goal was to check the outdoor temperature, and open the window if it is higher than inside the room.

Sprint 1 : Turn on/off the heating

First of all, before turning on/off the heating, we need to know the current temperature inside the room. To do this, we used a fake sensor implemented on OM2M. Under the container TemperatureSensor, we periodically declare a fake temperature value thanks to a Node Red server. If this value is under 20°c, heating will turn on. In other cases, it will turn off.

To retrieve this value, we created a service called Environment. This service will be deployed on port 8084. We created a function getTempInt able to retrieve the indoor temperature value stored on OM2M with an HTTP exchange. When the service is deployed, users are able to access this value with a GET method.



Attribute	Value
ty	4
ri	/mn-cse/cin-380364292
pi	/mn-cse/cnt-614764092
ct	20201224T173200
lt	20201224T173200
st	0
cnf	float
cs	5
con	20.06

Now that our inside temperature is available, we created another service called Controller, which will be our main Rest API. This service retrieves the temperature using a GET request on the Environment microservice, and stores it. The first version of our controller only stores the indoor temperature and subsequently turns the heating on or off. In order to simulate the heating system, we created a microservice called HeatingManagementMS that is in charge of managing the list of heaters and their state. It responds to GET requests, and models the heaters using a HashMap (code extract below).

```java
@RestController
@RequestMapping("/Heating")
public class HeatingResource {

    public static HashMap<String,Boolean> chauffagesMap;//Creating HashMap
    static {
        chauffagesMap = new HashMap<String,Boolean>();
        chauffagesMap.put("chauffageSalle1", true);
        chauffagesMap.put("chauffageSalle2", false);
        chauffagesMap.put("chauffageSalle3", false);
        chauffagesMap.put("chauffageSalle4", false);
    }

    @GetMapping("/setChauffage/{idStr}/{state}")
    public boolean setChauffage(@PathVariable String idStr, @PathVariable boolean state){
        chauffagesMap.replace(idStr, state);
        return true;
    }

    @GetMapping("/Reset")
    public boolean resetMap(){
        chauffagesMap.clear();
        return true;
    }
```

Sprint 2: Open/Close Window

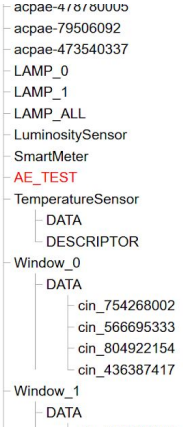
In this second sprint, we wanted to control the window state thanks to a rest service. To store each current window state, we are using an OM2M architecture too. We created two containers called Window_0 and Window_1, which will store windows states with a boolean value (true = open, false = close). To recover these values and to update them, we created another service (WindowManagementMS). We developed a java resource program which is composed of three functions.

The first one (getWindowState) allows users with a GET method to retrieve the window state called with his id (0 or 1). This function will use an HTTP exchange with OM2M to access the value.

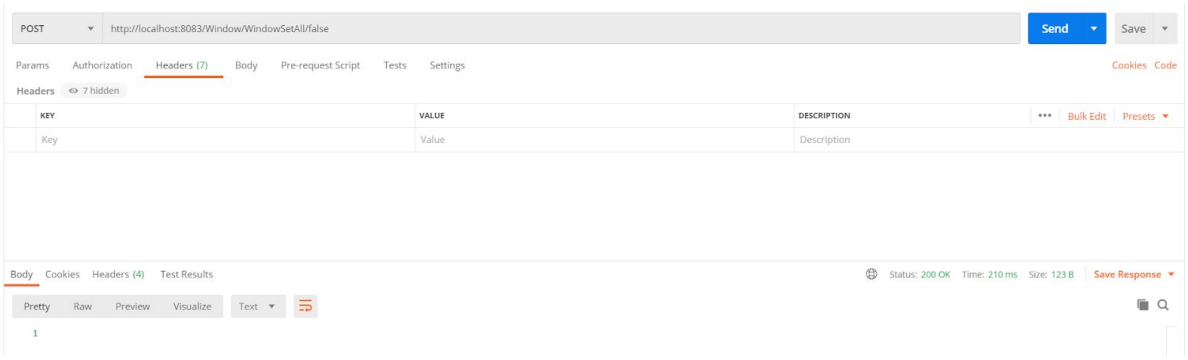
The second one (setWindow) allows users with a POST method to update the window state called with his id (0 or 1). This function will use an HTTP exchange with OM2M to update the value.

The last one (setWindowAll) is an optimization of the second one. Thanks to the function, users are able to set in the same state every window at the same time.

We are showing you below a case of use.



Window_0 and Window_1 are open (data= true).



We are sending a POST method to close all windows with Postman.

We are finally asking to know the window_0 state. This one is now closed, so our functions work well.

Sprint 3 : Temperature Management

Finally, we are able to close/open windows and to turn on/off heating and we want to create a specific organisation between windows and heating in order to save energy for the administrator. First application will be to check the outdoor temperature before turning on the heating. Main objective of this application will be to open the windows if the outdoor temperature is higher than inside, and below 20 degrees. The second application will be to assure that windows are closed before turning on the heating.
To realize the first application, we need to add a function to the micro service previously described: Environment. We developed the function called getTempOut, which we can call with a GET method and it will retrieve on the server www.prevision-meteo.ch, the current temperature in Toulouse. As you can see below, this function returns only a float representing the temperature.

Controller response

After continuous incremental improvements during all the sprints, we were able to develop a controller that is able to automatically change the windows and heaters according to inside and outside temperatures. The final version of the controller grabs the temperatures, the state of windows and the states of heaters. With these parameters, the controller makes decisions and changes the states of the equipment with the aim to maintain an inside temperature between 20°C and 25°C.

The possible cases considered are as follows :

T°in	T°out	Actions
20<T<25		Close windows (if open), turn off the heating (if on).
>25	>T°in	Turn off heating (if on). Close the windows.
	<T°in	Turn off heating (if on). Open the windows.
>25	<25	Turn off heating (if on). Open the windows.
<20	>20	Turn off heating (if on). Open the windows.
<20	<20	Turn on heating (if off). Close the windows.

Choices were made with the idea to save energy by only turning on the heating if necessary. Here are examples of the controller's responses in cases "T°in<20 & T°out<20" (left) and "T°in>25 & T°out<T°in" (right).

Controller Running

Current Window 0 state : Closed

Current Window 1 state : Open

Current Heater 1 state : On

Current T° outside : 2.0

Current T° inside : 1.56

We want the inside T° between 20° and 25°

Controller will turn on the heating and close the windows if they are open.

At least 1 window is open, the controller will close all the windows. It's cold outside !

State after modifications by the controller

Current Window 0 state : Closed

Current Window 1 state : Closed

Current Heater 1 state : On

Controller Running

Current Window 0 state : Closed

Current Window 1 state : Open

Current Heater 1 state : On

Current T° outside : 2.0

Current T° inside : 37.17

We want the inside T° between 20° and 25°

Controller will verify that heaters are off. It's too hot!

But it's better outside. Controller will open the windows.

At least 1 window is closed, the controller will open all the windows.

State after modifications by the controller

Current Window 0 state : Open

Current Window 1 state : Open

Current Heater 1 state : Off

Conclusion

During this project, we developed several Rest services, which allow users to manage the inside room temperature with an operation on windows, heating or both. To run this application, users need to send a request GET to the main service (controller) at his address. Currently, our application does nothing if users didn't send any request. We can imagine, as an evolution project, to develop an autonomisation of the service and to improve the controller in order to make it able to make finer decisions for more possible cases.