(Mini-)Project Report

Justification of the Application Architecture

As part of this project for the automatic management of a smart home, we designed a **modular architecture** based on several microservices, each responsible for a sensor or actuator (LED, light sensor, temperature, humidity, button, LCD display). These services communicate via **REST webservices** and, in some cases, via **MQTT** to meet the specific requirements of connected devices.

Choice of REST Architecture

We chose to structure our application around **REST webservices** for several reasons:

Simplicity and Universal Compatibility

- REST is based on the HTTP protocol, which is natively supported by the hardware used (ESP8266 via `ESP8266HTTPClient`), Spring Boot servers, web frontends, and testing tools like Postman.
- This facilitates **development, testing, and integration**.

Clear and Modular Decomposition

- Each Spring Boot service represents a component of the house: `Humidity Service`, `LED Service`, `Sunlight Service`, etc.
- This makes the application **easy to maintain** and **extensible**, enabling **independent deployment** of each service.

We managed to keep the code highly generic, requiring only two images to be built:

- backend-service
- component-service

Thus, with a single docker compose, it is very simple to launch the entire application.

Standardization

- REST follows standard conventions (`GET`, `POST`, `PUT`, `DELETE`), which makes API documentation, understanding, and usage in third-party or future projects much easier.

Adding the MQTT Broker: Technical Justification

We also decided to integrate an **MQTT broker (Mosquitto)** between the ESP and the backend. While this can introduce **some network complexity**, this choice is **fully justified in an IoT context**:

MQTT is Designed for Connected Devices

- The ESP is a **resource-constrained board**, and HTTP communication quickly reaches its limits (header size, network load, TCP connection management).
- MQTT is a **lightweight and optimized protocol**, based on TCP/IP, designed to operate in constrained environments.

Reduced Network Load

- Instead of sending multiple and simultaneous HTTP requests, the ESPs can simply publish their data to an **MQTT topic**, with minimal network overhead.
- The backend only needs to **subscribe once** to receive real-time data from multiple sensors.

Producer/Consumer Decoupling

- The publish/subscribe model provides **complete decoupling between the ESPs and the backend**. This allows for greater flexibility: sensors do not need to know the addresses of data consumers.
- This architecture **makes it easy to add new sensors or services** in the future without impacting the rest of the system.

Suited for Real-Time IoT

- For scenarios where sensors send continuous values, MQTT is **more efficient and responsive** than repeated REST requests (polling).

ESP side: How to use it

Board architecture:

- ESP8266
- Carrier Board
- Grove - Temperature & Humidity Sensor (DHT11)
- Grove - Chainable RGB LED
- Grove - 16x2 LCD
- Grove - Light Sensor
- Grove - Button

We defined two use cases:

- Manual -> Uncomment the `#define MANUAL` in `config.h`
- Automatic -> nothing to do, the server backend will handle everything and work by itself.

How to make the project work:

- Change all `MACROS` in `config.h` (`ip`, `backend-ip`, `wifi-ssid`, `wifi-password`...)
- Start the backend using: **For backend** (At root of homeManager):
  - `docker build -t backend-service:latest .`

  For services (At root of sensor-service):
  - `docker build -t component-service:latest .`

  For the app (At root of this directory):
  - `docker compose up -d`
- Prepare and flash the esp:
  - Open the `first_tests/first_tests.ino` in the `arduino ide`.

    - In the `arduino ide`, install the required libraries in your Arduino IDE (see below).
    - Plug the esp and flash it.
- Configure the ihm:
    - Go to node-red: `http://localhost:1880/`
    - Import the `flows.json` file.
    - Install the `node-red-dashboard` library.
    - Go to the node `esp12/+/HUMIDITY/+` and add a MQTT broker with your ip
    - Select that newly created server for all the `MQTT` nodes (humidity, luminosity and temperature)
    - Change the topic of the buttons `MQTT` nodes **harcoding** the ip of the esp.
    - Change the ip of all the `http requests` nodes.
    - Deploy your ihm and access it
- Enjoy your homeManager

Work Done:

For this third part, here is how the system should behave:

- Create a WebClient that sends requests to a server. (That will later be the backend of our system)
- Create a WebServer that will handle requests from the backend. These requests are:
    - Get the current temperature and humidity values.
    - Get the current luminosity value.
    - Get the current state of the button (pressed or not).
    - Switch on a led (or switch it off).
    - Display things on the LCD.
    - Change the threshold for the luminosity sensor.
    - Change the threshold for the humidity sensor.

The ESP will communicate with the backend via MQTT and HTTP depending on the use (frequent communication -> MQTT, non frequent -> HTTP)

All of those endpoints are implemented to handle more devices in the future. They all have identifiers to distinguish between different devices. (For the moment limited to one device of each type quoted above)

HTTP Endpoints:

- `/lcd/{id}/{line}` : Displays things on the LCD.
- `/led/{id}/{state}` : Switches on the led (or switches it off).
- `/humidity_threshold/{value}` : Changes the threshold for the humidity sensor.
- `/luminosity_threshold/{value}` : Changes the threshold for the luminosity sensor.

MQTT publications:

The esp8266 is publishing to topics (`esp12/+/+/+`). The luminosity, humidity and temperature sensors are publishing to different topics. (using the `PubSubClient` library)

Remarks:

Two alternatives were proposed for the implementation of a server on the ESP8266:

https://arduino-esp8266.readthedocs.io/en/latest/esp8266wifi/server-examples.html AND https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WebServer

Advantages of the first one:

- Already included in the ESP8266 core library, no need to install anything else.
- Already used for the Wifi module

Disadvantages of the first one:

- No endpoint creation possible, only one endpoint is available. (http://{ip}/)

Size comparison:  before  after

After this comparison, we decided to use the second library (ESP8266WebServer) for the server implementation. That allowed us to create endpoints properly and have a clean implementation.

How to use:

1. Install the required libraries in your Arduino IDE (see below).
2. Connect the devices to the carrier board.
3. Upload the code to your ESP8266 board.
4. Open the Serial Monitor to see the output from the temperature and humidity sensor.
5. Play with the sensors to see the full behaviour of the system.

Useful docs:

- Arduino ide
- Temperature and humidity sensor: https://wiki.seeedstudio.com/GroveTemperatureAndHumidity_Sensor
- LED : https://wiki.seeedstudio.com/Grove-Chainable_RGB_LED/
- LCD : https://wiki.seeedstudio.com/Grove-16x2_LCD_Series
- Luminosity sensor : please download the library available on moodle
- PubSubClient : Available on the arduino ide library manager

Faced issues:

- For the leds: We use the backend and microservices for the sake of the exercice. When we push a button, the backend is queries and says the led to switch on. This leads to extreme latency. When we push the button in automatic mode, it takes around 20 seconds to switch on/off the led. In real conditions, the button should directly switch on/off the led (like manual mode).

- For the leds again: We used a very generic way of calling the endpoints. The backend always need the ip of the esp to perform actions to it. When the esp itself asks for something, it works perfectly. When an other device wants to control it (ex: the ihm), it should know the ip of the esp. That is not clean and doesn't reflect reality but that was the only way to do it.

- For the threshold: It doesn't work in the ihm, the endpoint on the backend is not working properly. But all the logic is impemented and is worth looking into.

How do I Run it ?

Build based images

For backend (At root of homeManager):

- `docker build -t backend-service:latest .`

For services (At root of component-service):

- `docker build -t component-service:latest .`

For node-red (At root of node-red):

- `docker build -t node-red-custom:latest .`

For the app (At root of this directory):

- `docker compose up -d`

Where do I send my requests ?

You only need to perform requests to backend, `http://<machine:IP>:8080/ssse/sensor`:

-`/create`: with DTO:

```
public class ComponentDTO {
    private String id; //Must be ipaddress of the esp +
componentname/nbcomponent (for example 192.126.1.1.0temperature1)
    private ComponentType type;
    private String value;
    private String timestamp;
}

public enum ComponentType {
    Humidity = 0,
    Sunlight,
    Button,
    Temperature,
    LCD,
    LED
}
```

- `/update/threshold` : with DTO:

```
public class ThresholdDTO {
    ComponentType componentType;
    Integer treshold;
}
```

The rest of the communications are by MQTT throught `tcp://localhost:1883`: The backend listent on `esp12/+/+/+` where:

- `first +`: ip address of esp
- `second +`: sensork KEY {HUMIDITY, TEMPERATURE, SUNLIGHT, BUTTON}
- `third +`: sensor ID = esp ip address + sensorname/nbcomponent (for example 192.126.1.1.0temperature1)

ATTENTION, for the temperature and the sunlight sensors: the associated led id is led0 and for button it's led1 (fixed name).