INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Département de Génie Électrique & Informatique

Projet Integrateur 5^{eme} année

Vehicould - Monitoring our city

January 20, 2022

Students :

Florian	CONVERT	convert@etud.insa-toulouse.fr
Baptiste	LERAT	lerat@insa-toulouse.fr
Abir	BENAZZOUZ	abirbenazzouz@gmail.com
Ewan	MACKAY	mackay@insa-toulouse.fr
Gregoire	HEBRAS	gregoire.hebras@2021.icam.fr

Tutors :

Thierry	MONTEIL
Joseph	SHEA

Abstract :

Our world is getting more and more urbanized. In 2020, we count 4 billion people living in cities, but according to the WHO, one person out of five dies because of air pollution in cities [1]. Today, different real time air quality monitoring solutions are emerging on the web but we believe they treat the problem superficially. We want to provide decision makers with a tool allowing them to take decisions at the finest scale possible. We have imagined a solution that, thanks to mobile sensors embedded on public bikes, could collect data at any point of the city and point out very precisely which areas require special attention, notably with regards to atmospheric pollution. This report presents the development of a prototype that we want to deploy in the city of Toulouse, France. We provide every detail of the project, from the first drafts of a solution through the design phases to the results obtained with the prototype.

Keywords: Wireless Sensor Network; Internet of Things; Data Mapping; Air pollution ; Bluetooth Low Energy

Contents

Introduction

As engineers, the ecological transition is is a central issue for tommorow's society. The "Plan Climat Air Energie Territorial" (PCAET) published by Toulouse Metropole is a sustainable territorial project serving as a guideline to coordinate the fight against local pollution and the adaptation of the region to climate issues related to air quality. Along with decreasing air quality due to emissions, some studies led in partnership with MeteoFrance predict a global increase in temperature of +2 and +4°C by the year 2100. In urban areas, the effects of pollution will be strongly felt by inhabitants, seeing as urban areas are a concentrated area of emissions. Closer proximity to sources of emissions, on a daily basis, can deteriorate health of inhabitants and the livability of a city. As more and more people live in urban areas, this issue will only be exacerbated.

Structures such as buildings, roads, and other infrastructure absorb and re-emit the sun's heat more than natural landscapes such as forests and water bodies. In urban areas these structures are concentrated, implying less greenery, in turn gives rise to numerous heat islands in the cities. To overcome these problems exposed earlier, cities need to locate the heat islands and the areas of highest atmospheric pollution. Then they can deduce the origin of these disturbances and implement policy to tackle the problem. The idea is to identify all the causes of pollution hotspots. But with the means currently available to them, it is difficult for cities to follow precisely the variation of the climate parameters (temperature, humidity, gas density) hour by hour. For example, in Toulouse there are only nine IQAir stations that measure the air quality. To fight against pollution in an urban area such as Toulouse, it is absolutely essential to track these indicators.

Our idea is to implement a sensing device onto the city bikes which can measure the humidity, temperature and polluting gas density at regular intervals. Every time a citizen uses a bike it will take measurements all along his ride. When the ride ends, the user recovers the bike at the bike station and the data collected by the device is transmitted to the bike station. In turn, the station sends the data to an online database. Using a map interface displaying data with the chosen parameters (time, temperature, gas, etc), the local authorities can easily and precisely oversee the location of heat island and the pollution hotspots. With 283 city bikes stations in Toulouse, the flow of people using the bikes is important enough to grid the city and collect a significant quantity of data to make relevant hypotheses on the origin of air pollution. The city can target the critical areas and reflect on the urban improvements needed to reduce this pollution, making the life of inhabitants better.

What we wish to provide is a tool for better understanding atmospheric pollution in urban areas. By better visualizing the phenomenon, we believe effective decisions can be taken on the local scale.

1 Existing solutions and comparison of air quality monitoring systems

In this part, we aim to provide an overview of existing air quality measurement solutions. We will focus on the IoT aspects but also explain how most of the solutions come with a mathematical model enabling real-time measurements and predictions.

1.1 Traditional air quality measure with static stations

In the past several years, more and more consumer applications give a visualization of air pollution in cities via the AQI (Air Quality Index) and thanks to static stations implemented around the city. For example in Toulouse, Atmo gives very precise data on air pollution at the locations of their static stations.

The article [2] gives an example of this type of station working with a Rest API and giving data for cities in Romania. They use both wired and wireless sensors (BLE/ZigBee) and the data is published using the MQTT protocol through a Mosquitto broker.

If we want to provide a visualization of data for the entire city we need to implement mathematical models in our architecture. Indeed as explained in [2] thanks to weather models and weather forecasts we can extend the data collected to several zones so that they cover a much greater zone and give a heat-map of the entire city. We believe that this solution which is used in many use cases does not provide very precise data at any given location since it can not consider every source of pollution in a city in their mathematical models. The idea of our project is to identify precise points in a city where pollution requires attention.

1.2 Crowd-sourced solutions

Another solution for air quality monitoring applications which are commonly used today are crowd-sourced solutions such as IQ'Air and Pluma. The idea behind the two brands is the same : give anyone the opportunity to collaborate with his/her own station. However, there is a difference between the two solutions : IQ'Air provides a static station and Pluma provides portable stations so that people can measure air quality during their commutes and publish them afterwards.

These solutions work fairly well in some cities where people are aware of this issue such as in Amsterdam. Nevertheless, in order to help the authorities to make accurate decisions in their cities, it is important that they are the one controlling and powering the application with their own infrastructures. This would prevent the users from using their own devices, for example Pluma application uses the user's smartphone and so, his personal data which causes ethical and security concerns. The city also needs to have confidence in the authenticity and integrity of the data, and depending on individuals for data inflow is less reliable.

1.3 Air monitoring with IoT in city buses

The article [3] from a research project in Sweden gives a very interesting solution to monitor urban pollution data using sensors on city buses. The research that they conducted confirms our opinion that moving sensors provide a more cost-efficient coverage of the city. They also share our idea that this type of solution should be embedded in public transport because it should be the local authorities that manage this type of project. The Swedish team decided to put sensors for collecting CO, NO2, temperature, humidity and pressure on top of the city buses. The sensors communicate to a RestAPI via HTTP Post request through the 4G cellular network.

Research shows that deploying sensors on city buses has many benefits : moving sensors, big rechargeable battery (less energy problems) and hardware security to prevent physical damage.

Nevertheless the fact that buses always follow the same trajectory creates significant lack of data and we believe that the same idea on bikes is much more interesting in terms of diversity of data origin.

1.4 Sensors for pollution on bikes

Finally articles [4] and [5] provide an idea for embedded sensors on bikes, which happens to be the solution we adopted. As mentioned in [5], moving sensors are the future of very fine scale urban pollution monitoring. The paper shows with statistical spatialization methods that it is possible to cover a town with moving sensors on bikes. [4] proposes a hardware solution and a network architecture using the user's smartphone as a gateway.

The different solutions presented gave us different points of view allowing us to consolidate our idea : research exists for mobile sensors on bikes and also the use of public transport for an IoT solution. This is why we believe that the best solution to create a tool for urban planners and decision makers helping them to see the finest analysis of their city is to create a network of sensors based on free-access bikes relaying the data thanks to gateways located on bike stations.

2 Solution for collecting data with a wireless sensor network (WSN)

In this part, we will explain how we want to use the existing shared-access bike network deployed to create a WSN covering the city thanks to predictable node mobility.

2.1 Architecture of the WSN

As we said, we want to collect data at any point of the city, not just generate a tendency based on a few point and a mathematical model. This is why we need moving sensors which is the best solution in term of cost-efficiency. Our application is dedicated to municipalities and urban planners, therefore, we think that the sensor should be embedded in public transports such has bikes or buses. We have chosen the bike cause we think the trajectories will give us a better set of data. By considering we embed the sensor on a bike we had to think how we wanted to transmit the data to our web server using a gateway. The architecture we imagined for this IoT application is described in the figure bellow.

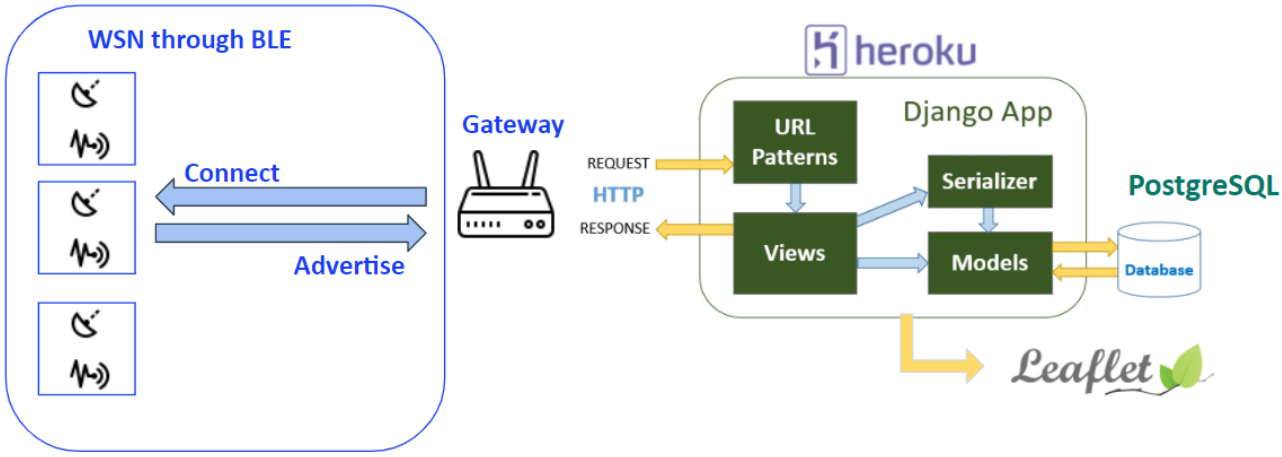


Figure 1: Architecture of our IoT Application

In next parts we will explain the different technical choices we used (BLE Protocol, RestAPI with Django, SQL Database).

The structure of a classic Wireless Sensor Network for IoT is composed of sensor nodes relaying data to a gateway which will send it the application's high-level server. For this project we have chosen this type of architecture, with the sensor node on a bike and a gateway located at a strategic point, making sure all bikes will transmit data: the shared-access bike station. With this methodology, even with moving nodes like in our case we can assume that every bike will be close to a gateway (<20m) regularly. With the previous hypothesis we have chosen the Bluetooth Low Energy (BLE) protocol which is the most efficient protocol for short range applications and power consumption. Indeed BLE is a protocol with a maximal theoretical data rate of 1Mb/s at a range of 10 meters with the best energy efficiency. The energy saving nature of this solution is only true for the sensor, as indeed, the gateway needs to constantly scan for devices it wants to connect to. This is why it requires more energy and needs to be in a fixed place like the bike station, providing enough energy to power it.

The gateway is also connected to internet, using the city free-wifi that allows us to forward the data to our API at a bit rate of 10Mbps. To summarise, we have different BLE servers which are mobile on our bikes and several BLE centrals, our gateways, that try to connect to them at any moment. With this method we will cover the entire city using fewer sensors than if they were stationary, every part of the city were the bikes are moving will be compared to other data and we will be able to identify very precise points where pollution is a problem.

2.2 Hardware components of the node sensor

Our node sensor is managed thanks to an ESP32 micro-controller which allow us to easily coordinate our different sensors and send data with Bluetooth Low Energy. We connect a GPS, one gas sensor and one temperature sensor to this board:

- GPS : Adafruit Feather Ultimate GPS is a low cost GPS integrating the component MTK3339 (5 meters precision). Thanks to the Adafruit library we are able to locate and time our data via a UART connection (TX/RX)
- Gas sensor : Grove Multichannel Gas Sensor by Seeed integrating a MiCS6814. This sensor allows us to collect data from 6 different gases. With the library we manage, via the I2C module, to get the ppm of the gas at any moment.
- Temperature and humidity sensor : Grove DHT11 by Seeed, with digital interface managed by its own library.

Additionally, we use the internal EEPROM of the ESP to store our sensor's data before we reach a gateway, positioned at a bike station. It is possible because we believe that most of the trajectories will last around 30 minutes since the first 30 minutes are free of charge for users These 4 major components allow us to collect all the data we need when the bike is moving and transmit it using BLE when the bike is in range of a gateway.

2.3 Explanation of the firmware

The aim of the firmware on the ESP32 is to start collecting data at the beginning of a bike trip, then start advertising whenever it stops : this will save energy. Then if a gateway tries to connect while the device is advertising, it will then notify (a way of sending data in BLE) all the stored data on the right characteristic. The following state diagram represents our vision of the firmware.

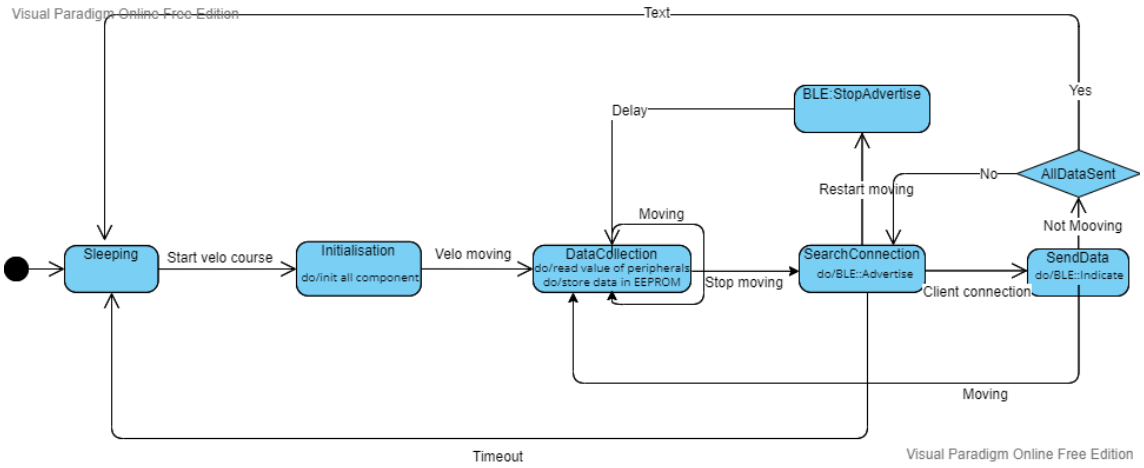


Figure 2: State Diagram of the ESP32 Firmware

The state diagram is working thanks to the following parts of the code. The first part of every Arduino code is the setup function. Here, we initialize all the drivers of our components and setup all the options we want:

```
void setup() {

  Serial.begin(115200);//Init UART
  DHT_sensor.begin();//Init DHT
  //Serial.println("Temperature sensor initialized");

  //Initializing gas sensor
  MiCS6814.begin();
  Serial.println("Calibrating");
  MiCS6814.calibrate();
  Serial.println("Calibrated");
```

Figure 3: Initializing sensors

This first step of our setup function initializes temperature sensors (with Analog input setup in the object declaration) and the gas sensor(correct pins I2C are declared in the object declaration) by calling the "begin" method from their library. The gas sensor can also be calibrated, it needs to be in the environment conditions indicated in the library.

```
 // 9600 NMEA is the default baud rate for Adafruit MTK GPS's- some use 4800
GPS.begin(9600);

// uncomment this line to turn on RMC (recommended minimum) and GGA (fix data) including altitude
//GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
// uncomment this line to turn on only the "minimum recommended" data
GPS.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCONLY);
// For parsing data, we don't suggest using anything but either RMC only or RMC+GGA since
// the parser doesn't care about other sentences at this time

// Set the update rate
GPS.sendCommand(PMTK_SET_NMEA_UPDATE_100_MILLIHERTZ);   // Update every 10 seconds to be sure we get at least 1 update in 20 seconds
// For the parsing code to work nicely and have time to sort thru the data, and
// print it out we don't suggest using anything higher than 1 Hz

// Request updates on antenna status, comment out to keep quiet
GPS.sendCommand(PGCMD_ANTENNA);
```

Figure 4: Initializing GPS module

Next, we need to initialize our GPS driver as shown in figure 4. The first line indicates the baud rate used to communicate through the Software Serial connection we created. Then, we transmit several pre-defined messages in the library to setup the parameters of our GPS:

- **PMTK SET NMEA OUTPUT RMCONLY** is a command asking the GPS only to get the minimal information : latitude, longitude and date provided by the RMC module. It won't use the GGA module determining the altitude for example.

- **PMTK SET NMEA UPDATE 100 MILLIHERTZ** set the frequency of GPS updates.
- **PGCMD ANTENNA** enable the GPS to transmit information about his status, it is necessary to be sure GPS see enough satellites to give us a correct position.

The last aspect we need to configure in the setup function is the BLE server.

```
  // Create the BLE Device
BLEDevice::init("Vehicloud_V01");

//Serial.println("Initializing BLE");
// Create the BLE Server
pServer = BLEDevice::createServer();
pServer->setCallbacks(&MyCallbacks);

// Create the BLE Service
BLEService *pService = pServer->createService(SERVICE_UUID);
CharGPS = pService->createCharacteristic(CHAR_GPS_UUID, BLECharacteristic::PROPERTY_READ|BLECharacteristic::PROPERTY_WRITE|BLECharacteristic::PROPERTY_NOTIFY|BLECharacteristic::PROPER
CharTime = pService->createCharacteristic(CHAR_TIME_UUID, BLECharacteristic::PROPERTY_READ|BLECharacteristic::PROPERTY_WRITE|BLECharacteristic::PROPERTY_NOTIFY|BLECharacteristic::PROP

  // https://www.bluetooth.com/specifications/gatt/viewer?attributeXmlFile=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml
// Create a BLE Descriptor
CharGPS->addDescriptor(new BLE2902());
CharTime->addDescriptor(new BLE2902());

// Start the service
pService->start();

// Start advertising
BLEAdvertising *pAdvertising = BLEDevice::getAdvertising();
pAdvertising->addServiceUUID(SERVICE_UUID);
pAdvertising->setScanResponse(false);
pAdvertising->setMinPreferred(0x0);  // set value to 0x00 to not advertise this parameter
//Serial.println("Waiting a client connection to notify...");
```

Figure 5: Initializing BLE

In figure 5 we have the different steps to setup a BLE server that has a characteristic in which we can send data, it corresponds to the following schematic :

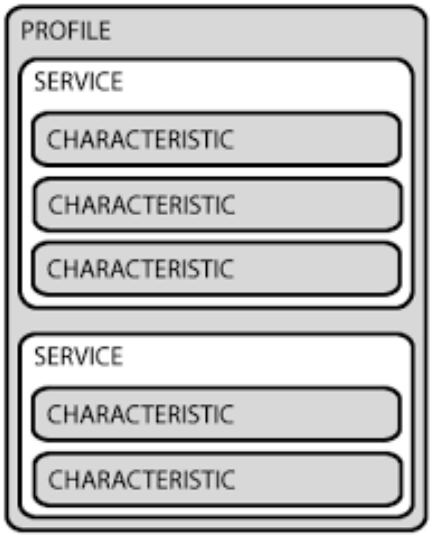


Figure 6: Example of BLE Server structure

Every service can have different characteristics, the server device provides services and characteristics and any device can subscribe to it in order to receive the data written on it like if it was billboard. In our case, the ESP32 provides a service with one characteristic in which we can send a maximum payload of 20 bytes. Now that all our components are correctly initialized we just need to correctly collect the data and manage when to send our buffers in the main loop of the Arduino code.

```
// so be very wary if using OUTPUT_ALLDATA and trytng to print out data
if (!GPS.parse(GPS.lastNMEA()))   // this also sets the newNMEAreceived() flag to false
  return;  // we can fail to parse a sentence in which case we should just wait for another
}

  if (millis() - timer > 20000) { //We take values every 20 seconds

   // float distance=acos(sin(radians(latitude_values[sample_number]))*sin(radians(GPS.latitudeDegrees))+cos(radians(latitude_values[s

    timer = millis(); // reset the timer
    Serial.print("Latitude:");Serial.println(GPS.latitudeDegrees);
    Serial.print("Longitude:"); Serial.println(GPS.longitudeDegrees);
    Serial.print("Hour:"); Serial.println(GPS.hour);
    Serial.print("Minute:"); Serial.println(GPS.minute);
    latitude_values[sample_number]=(GPS.latitude - 100.0f * int(GPS.latitude / 100.0f)) / 60.0f;
    latitude_values[sample_number]+=(GPS.latitude/100.0f);
    longitude_values[sample_number]=(GPS.longitude - 100.0f * int(GPS.longitude/100.0f))/60.0f;
    longitude_values[sample_number]+=(GPS.longitude/100.0f);
    Serial.print("Longitude:");Serial.println(longitude_values[sample_number]*100.0f);
    Serial.print("Latitude");Serial.println(latitude_values[sample_number]*100.0f);
    hour_values[sample_number]=GPS.hour;
    minute_values[sample_number]=GPS.minute;

    gas_values1[sample_number]=MiCS6814.get(CO);
    Serial.print("gas_values: CO");Serial.println(gas_values1[sample_number]);
    gas_values2[sample_number]=MiCS6814.get(NO2);
    Serial.print("gas_values: NO2");Serial.println(gas_values2[sample_number]);
    gas_values3[sample_number]=MiCS6814.get(NH3);
    Serial.print("gas_values: NH3");Serial.println(gas_values3[sample_number]);
```

Figure 7: Collect and store data

Figure 7 shows how we store data collected in different buffers (stored in the internal EEPROM of the ESP) every 20 seconds. The buffer size is limited to 200 units so we can store more than an hours worth of data. Once the data is stored, and since we start advertising when the bike stops, we check for connection thanks to the class BLECallBacks(Annexe) and send the data if the central is connected :

```
// Bluetooh
if (MyCallbacks.deviceIsConnected()&&sample_number>0){
  Serial.println("device connected, sending data");
  for (int i=0; i<sample_number;i++){
    //Serial.print("Latitude send :");Serial.println(latitude_values[i]);
    uint16_t Data_To_Send[10]={(uint16_t)((latitude_values[i]*100)), (uint16_t)(longitude_values[i]*100), (uint16_t)(hour_values[i]),(uint16_t)(minute_values[i]),(uint16_t)(temperatur
    uint8_t Data_BLE[20];
    tab_to_send(Data_To_Send, Data_BLE);
    //Serial.print("Sending sample number: ");Serial.println(i);
    CharTime->setValue(Data_BLE, sizeof(Data_BLE));
    CharTime->indicate();
    delay(3);
  }
  uint8_t Data_BLE[20]={0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255,0,255};
  CharTime->setValue(Data_BLE, sizeof(Data_BLE));
  CharTime->indicate();
  sample_number=0;
}
if (!MyCallbacks.deviceIsConnected()&& oldDeviceConnected){
  delay(500); //give the bluetooth stack the chance to get things ready
  pServer->startAdvertising();
  //Serial.println("Restart advertising");
  oldDeviceConnected=MyCallbacks.deviceIsConnected();
}
if (MyCallbacks.deviceIsConnected() && !oldDeviceConnected){
  //Reconnection
  oldDeviceConnected = MyCallbacks.deviceIsConnected();
}
```

Figure 8: BLECallBacks different state

This figure **??** shows how MyCallbacks permits us to handle the firmware state thanks to the interruptions sent by the BLE communication, indeed we can enter in different state after connection or disconnection.

We see in 10 that when we have a first connection we flush all the data in the same BLE characteristic. The frame has the following format :

uint16_t Tab

Latitude	Longitude	Heures	Minutes	Temp	Humidity	Gas1	Gas2	Gas3	BikeID
0	1	2	3	4	5	6	7	8	9

uint8_t Tab for BLE

Latitude LSB|MSB	Longitude LSB|MSB	Heures LSB|MSB	Minutes LSB|MSB	Temp LSB | MSB	Humidity LSB|MSB	Gas1 LSB|MSB	Gas2 LSB|MSB	Gas3 LSB|MSB	BikeID LSB|MSB
0|1	2|3	4|5	6|7	8|9	10|11	12|13	14|15	16|17	18|19

Figure 9: BLE Trame Format

Since the ESP32 BLE library needs uint8_t buffers to send data we had to create the following function to convert all our data :

```
void tab_to_send(uint16_t * tab_input, uint8_t * tab_output, int size_in=10){

  for(int i=0; i<size_in; i++){
    tab_output[(i*2)+1]=(uint8_t)(tab_input[i]);
    tab_output[(i*2)]=(uint8_t)(tab_input[i]>>8);
  }
}
```

Figure 10: Change uint16_t table to uint8_t

To conclude, thanks to a system handling different states in function of the BLE connection and bike velocity, we manage to have an autonomous device collecting data when bike moving and sending it when he stops at a station.

2.4 The gateway

The gateway has to collect data from the ESP32 via Bluetooth Low Energy. Data needs to be processed to be stored and used directly on the gateway and be sent to the database. We use a Raspberry Pi 4 B+ which has a BLE device and permits us to create a full meteorological station.

2.5 Firmware on the gateway

The Raspberry Pi OS is installed on the Raspberry Pi. This operating system has the advantage of being lightweight and fast. It does not provide any graphic interface but it is not necessary for our use.

The architecture of the firmware is composed as follows:

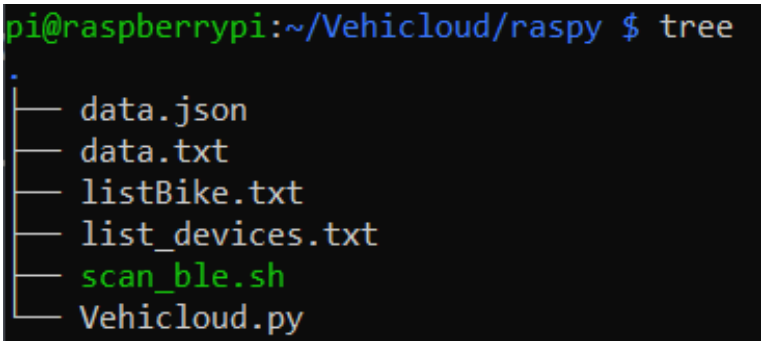


Figure 11: Architecture of the firmware

data.json : stores all the json files generated as a dictionary
data.txt : stores the data as text
list_devices.txt : stores the response of the scan_ble.sh file
listBike.txt : stores all the mac addresses of the bikes on the network
scan_ble.sh : Scans all the BLE devices near the gateway and stocks the response into the list_devices.txt file

Vechicloud.py is a Python application running on the gateway. We call different libraries : BlueZ, pexpect, time, request, json, os and datetime.
The first step is to detect if a bike is close to the station and then connect to it.
This is the role of the scan_ble.sh script. It exectute the command *sudo hcitool lescan* which is a command of the native Linux kernel library BlueZ. The script scans all the BLE devices near the platform. Un example of the response saved into the list_devices.txt file is shown below

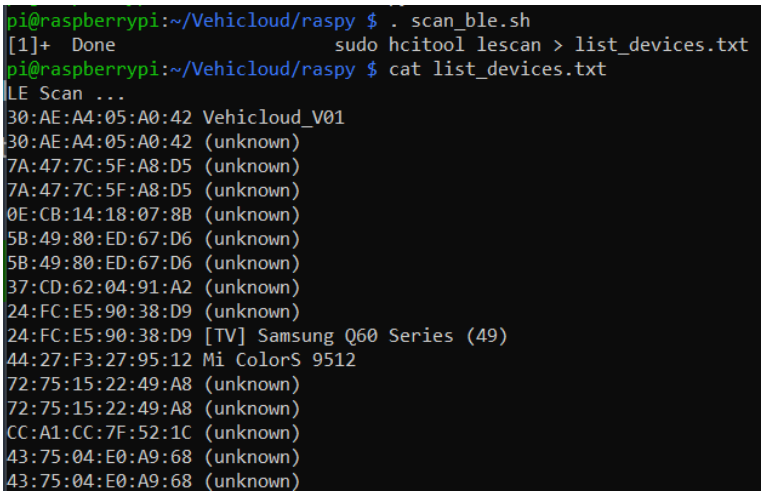


Figure 12: Example of a scan_ble.sh script response

All the bikes deployed on the network are named Vehicloud_XXX. We need to detect a device whose name begins name begin by Vehicloud, then we save its MAC address and connect to it. Then we extract only the MAC address for the devices called VehicloudXXXX and stock them into the file listBikes. We browse the file and connect to a bike with the gatttool library.

```
13 # Run gatttool interactively.
14 print("Running gatttool...")
15 child = pexpect.spawn("gatttool -I")
16
17 # Read bike file
18 bikeFile = open('listBike.txt', 'r')
19 for bike in bikeFile:
20
21     print("Bike address:"),
22     print(bike)
23 # Connect to the device.
24     print("Connecting to "),
25     print(bike)
26     child.sendline("connect {0}".format(bike))
```

Figure 13: connexion step

Once the connection is made, we begin to collect data from the bike with the command: char-write-req 0x2e 0200. It indicates that we want to read data in the indicated characteristic.

```
while receiveData :
    child.sendline("char-write-req 0x2e 0200")
    child.expect("Indication   handle = 0x002d value: ", timeout=10)
    child.expect("\r\n", timeout=10)
    data = child.before.decode('utf-8')
    with open('data.txt', 'a') as f: f.write(data+"\n")
    data = data.split(" ")
```

Figure 14: Collecting data

Then we extract a string of 20 bytes in length. We separate each byte and store them in a JSON file with the correct syntax that is defined at the API level. All the JSON files are stored in a file in the Raspberry Pi in order to be used directly on the gateway to be used as a weather station for example. We collect data until we receive the EOF flag which announces the end of transmitting information. The EOF flag is equal to :

EOF = [’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’00’,’ff’,’’]

```
if(data == EOF):
    receiveData = False
else:
    bikeId = int(data[8],16)
    time = str(myDate.year) + "-" + "0"+str(myDate.month) + "-" + "0"+str(myDate.day) ,
    + "T" + str(int(data[5],16)) + ":" + str(int(data[6],16))+":"+ str(int(data[7],16)) + "Z"
    location_lat = str(int(data[0],16)) + str(int(data[1],16))
    location_lon = str(int(data[2],16)) + str(int(data[3],16))
    temperature = str(int(str(int(data[10],16)) + str(int(data[11],16)))/100)
    humidity = str(int(data[12],16)) + str(int(data[13],16))
    gaz1 = str(int(data[14],16)) + str(int(data[15],16))
    gaz2 = str(int(data[16],16)) + str(int(data[17],16))
    print(bikeId),print(time),print(location_lat),print(location_lon),print(temperature),
    print(humidity),print(gaz1),print(gaz2)
    json_file = {"bikeId":bikeId, "time": time, "location_lat" : location_lat, "location_lon":location_lon,
    "temperature": temperature, "humidity" : humidity, "gaz1": gaz1, "gaz2": gaz2}
```

Figure 15: Formating data

We are now able to send the data with a post request to the API.

```
response = requests.post('http://vehicloud.herokuapp.com/sensorDatas/',json = json_file )
print(response.text)
print(response)
print(".")
```

Figure 16: Sending data to the API

To sum up here is a flowchart that explain the program.

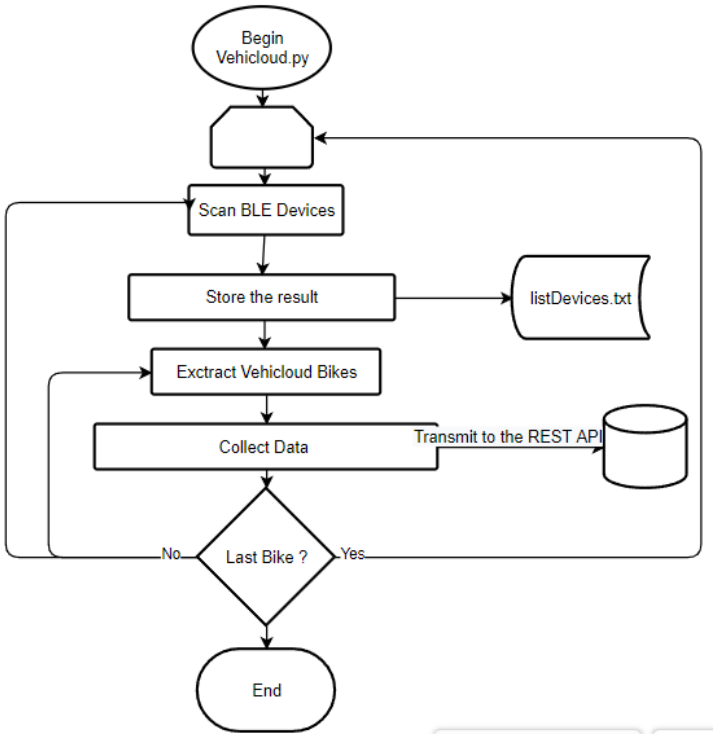


Figure 17: flowchart of Vehicloud.py

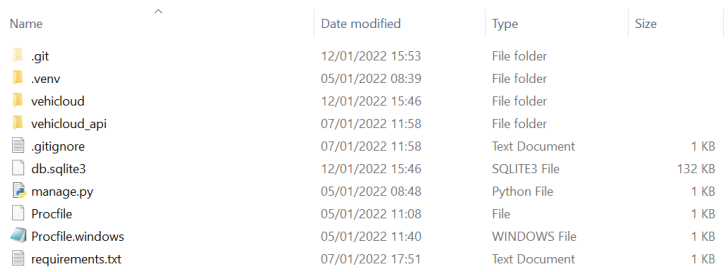
3 Strategy for storing data and presenting it on the web

By creating data on our bicycle borne device and by sending it to our gateway, we effectively unload the data from local storage. The gateway not only serves as the connection to the internet, to send this data onto the internet for further processing and display, but also presents the possibility of filtering and pre-processing of data before sending it off. To be able to host data online, our solution evolved through multiple iterations, using different existing tools such as Node-RED and the OneM2M standard, all the way to the solution retained, creating an API by hosting a Django REST framework on the Heroku platform, linked to a PostgreSQL database and interfacing with Leaflet to create an interactive cartography. We detail this process below.

3.1 Developping a Django RESTful framework

The first link in this chain is to create an API to be able to have somewhere to send our sensor data. The idea of deploying an API came to us as the logical evolution once we realised that Node-Red was not adapted for our solution as it is too limited. We needed a way to verify, filter and store data remotely, and Node-Red was not adapted to these tasks. We need to have a fully functional API that can be the interface between different components, such as our database and our visualisation solution. This API is the middleman between our gateway and our online application.

To do so, we implement the Django REST framework, which is a powerful and flexible toolkit for building Web APIs. In other terms, the app we develop with the Django framework is the software component that describes how we send data online. We have chosen this solution due to its ease of implementation, speeding up the deployment process by cutting development time. It is also particularly scalable, seeing as how some of the busiest sites on the web leverage Django's ability to quickly and flexibly scale. The Django framework is written in Python and streamlines much of the web development process. Every file in the Django application directory is needed. The manage.py file is the file that will eventually run the other files in the vehicloud and vehicloud_api folders.



Name	Date modified	Type	Size
.git	12/01/2022 15:53	File folder	
.venv	05/01/2022 08:39	File folder	
vehicloud	12/01/2022 15:46	File folder	
vehicloud_api	07/01/2022 11:58	File folder	
.gitignore	07/01/2022 11:58	Text Document	1 KB
db.sqlite3	12/01/2022 15:46	SQLITE3 File	132 KB
manage.py	05/01/2022 08:48	Python File	1 KB
Procfile	05/01/2022 11:08	File	1 KB
Procfile.windows	05/01/2022 11:40	WINDOWS File	1 KB
requirements.txt	07/01/2022 17:51	Text Document	1 KB

Figure 18: Django Application file hierarchy

It is important to note that the hidden .venv file is automatically generated when we created our project in an isolated virtual environment. Virtual environments manage separate package installations for different projects. They essentially allow us to create a "virtual" isolated Python installation and install packages into that virtual installation. When we switch projects, we can simply create a new virtual environment and do not have to worry about breaking the packages installed in the other environments. It is always recommended to use a virtual environment while developing Python applications. This way, we don't have to worry about differing dependency installations, as all the modules needed are listed in the requirements.txt file.

Our Django application is seperated into two different files. All the files in both directories are essential for the application to work. Lets have a look inside them.

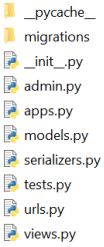


Figure 19: Contents of the vehicloud file

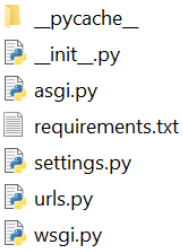


Figure 20: Contents of the vehicloud_api file

Notable files in the vehicloud file are :

- The models.py describes the name of a data field passed in a transmission to the API, it's type of data (such as integer value or float) as well as it's number of digits and decimal places. This allows us to already filter out any impossible values. For example a temperature reading of 32.1187838°C will not be accepted and the message will not be saved, same for a reading of 1863.0°C. It is this file that will affect the format of data to be sent.
- The serializers.py file describes the order of fields of data used by the serializer to connect data to the previously created model. As a reminder, serialization is the process of converting an object into stream of bytes so that it can be transferred over a network or stored in a persistent storage.
- The urls.py file is the file that sets up the url structure of our app. We have set it up in this project to have automatic URL routing by using a package within Django's rest framework.

Notable files in the vehicloud_api file are :

- The settings.py file is arguably the most important file in the entire app, as it is this file that sets many of the app's parameters. Defined within are the acceptable hosts, database types, middleware imports, security keys and other important configurations.
- The wsgi.py file is a necessary file for deploying the application. Django's primary deployment platform is WSGI, the Python standard for web servers and applications. WSGI (or Web Server Gateway Interface) is a specification that describes how a web server communicates with web applications, and how web applications can be chained together to process one request. Seeing as how WSGI is a standard, to actually deploy we will be using Gunicorn, a pure-Python WSGI server for UNIX.

With this application running locally, we can post data frames to the app by using an HTTP verb to a specific URL. Let us first look at the developed application running locally.

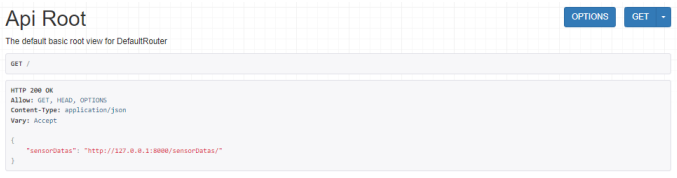


Figure 21: Django app running locally with URL : https://localhost:8000/

As we can see, we have set up hyperlink and automatic URL routing to allow navigation within the application. From this home page, lets navigate to where the data influx is visualised.

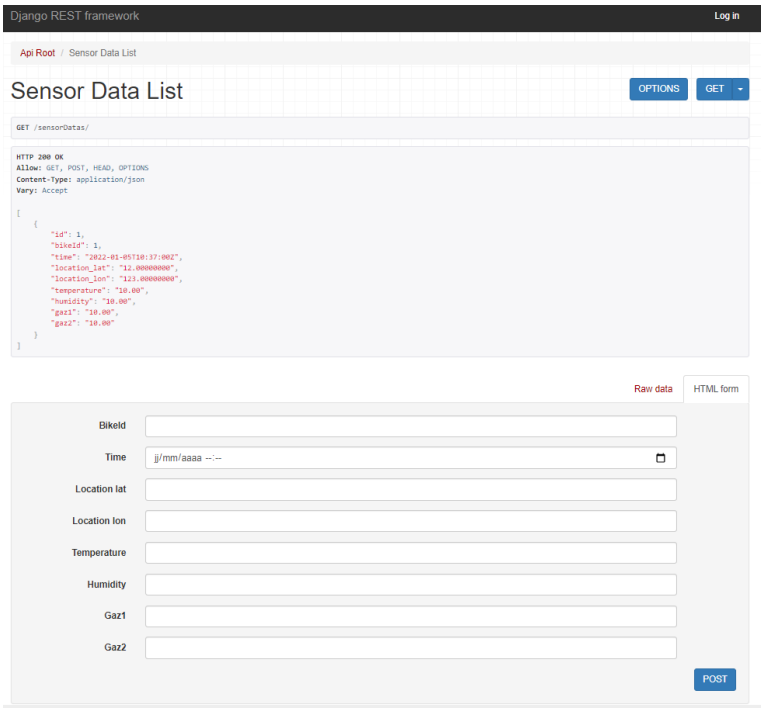


Figure 22: Django app listing all the sensor data messages it received locally with URL : https://localhost:8000/sensorDatas/

In the figure above, we can see different characteristics of our application. Our data is presented as a JSON containing different data fields, indexed by a field name (temperature, latitude...etc). Every message incoming on this application is indexed by a unique, automatically generated id field, that allows us to extract a specific message and it's data if needed. We have included an HTTP post form, that allows us to simulate different messages as a way to test our different integrations, such as the database. We can also post data to this application by using Postman, a free software that allows us to send HTTP requests easily.

We have set up the application to have three types of pages :

- https://localhost:8000/ : our home page. It does not serve any real purpose except for redirection.
- https://localhost:8000/admin/ : our administration page, automatically generated using Django's RESTful bootstrap, accessible only to the superuser we have created.
- https://localhost:8000/sensorDatas/ : list of all our data messages, indexed by a unique integer identifier. If we delete a message, all our data identifiers change dynamically.
- https://localhost:8000/sensorDatas/id/ : retrieval of a unique message and it's data for a de-cluttered view of a single JSON containing all of a messages information.

We now have an application running locally with a particular URL, to which we can send JSON data frames using HTTP requests, and this application will identify them and store them. In the local deployment of our app, we use the SQLite database provided by Django at the inception of our app. This database is adapted to local development but is not persistent. That is to say that when we terminate the app, we delete all the data that was stored during that instance. To solve this issue, our decision was two-fold : make the app accessible online and host the database online.

3.2 Deploying the Django app using Heroku

The issue we faced at this stage is how to deploy our application to the web so that we have a publicly available domain we can send data to. We needed a hosting platform that could execute our Django application and store data remotely by interfacing with a database. We chose Heroku, mostly because it is the most widely used platform to deploy Django apps, so there was lots of documentation available to help in our deployment. Heroku is a container-based cloud Platform as a Service (PaaS). Applications that are run on Heroku typically have a unique domain used to route HTTP requests to the correct application container or "dyno". This is very practical for our project. Heroku allows us to deploy, manage, and scale our app, with minimal effort.

To deploy onto Heroku, we must link our already existing git version control software, containing our Django app, with Heroku. Indeed, pushing our code to a remote branch linked to Heroku will run the code contained in the pushed repository at the specified URL. This way we create an app on Heroku named Vehicloud and we choose the URL : vehicloud.herokuapp.com. Later on, once the deployment is successful, our app will run at this address and any HTTP methods are targeted to this domain name.

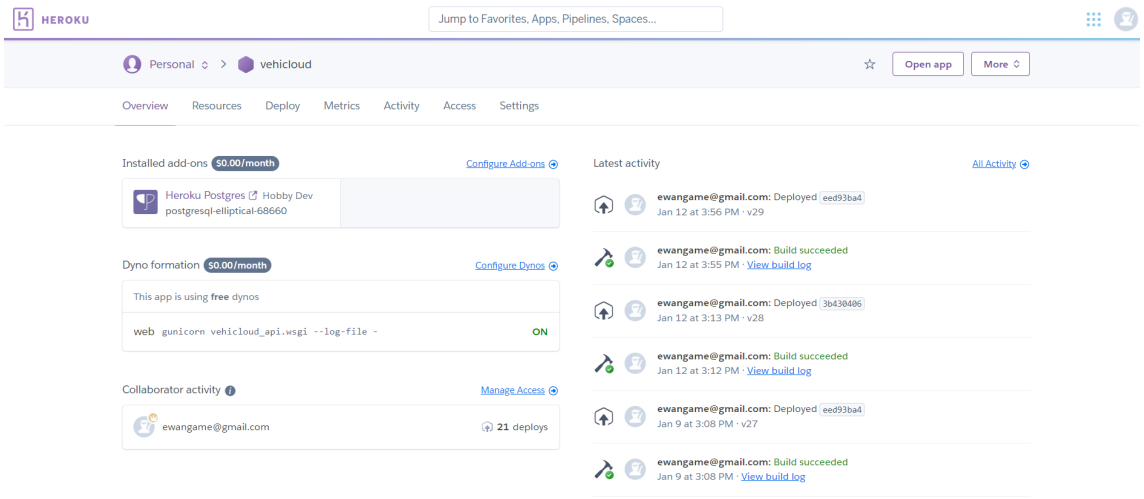


Figure 23: Heroku Application created

From the Heroku dashboard we can control many different aspects of our app deployment, notably the associated database. Since we do not plan to show our API to any end users, we don't need to create a particularly good-looking website for our application, as long as it works.

With our Django app deployed to the web, the gateway can now post data in a JSON format using an HTTP request to the URL : vehicloud.herokuapp.com. The data is now available remotely, and is accessible from any device with an internet connection. The Heroku application has the same URL structure as the local application, as only the domain name changes.

3.3 Data storage and accessibility

Seeing as how data inflow will grow, and that we needed our data to be persistent (not deleted during any downtime), we needed to change our database. Indeed, if no activity is detected on the application for 30 minutes, it goes to sleep. Our data would not survive this downtime using the standard SQLite database.

Over the last few years, two open source database servers have become dominant on the Web. These servers are MySQL and PostgreSQL. In order to provide the best service available, Heroku decided to use PostgreSQL, and provide this service as the default choice for all database hosting. PostgreSQL is an open system and always will be (unlike MySQL). This means that now and moving forward, as long as we are using PostgreSQL for our database server, you will not be subject to any vendor lock-in. Thus, as a user, we are able to take your data wherever we please. This is critical to our use of Leaflet for our front end. There are a number of other technical reasons as to why PostgreSQL is the favored option, too: for instance, transactional data definition languages (DDLs), fast index creation with concurrent indexes, extensibility, and partial indexing and constraints. For us however, the main criteria was ease of implementation.

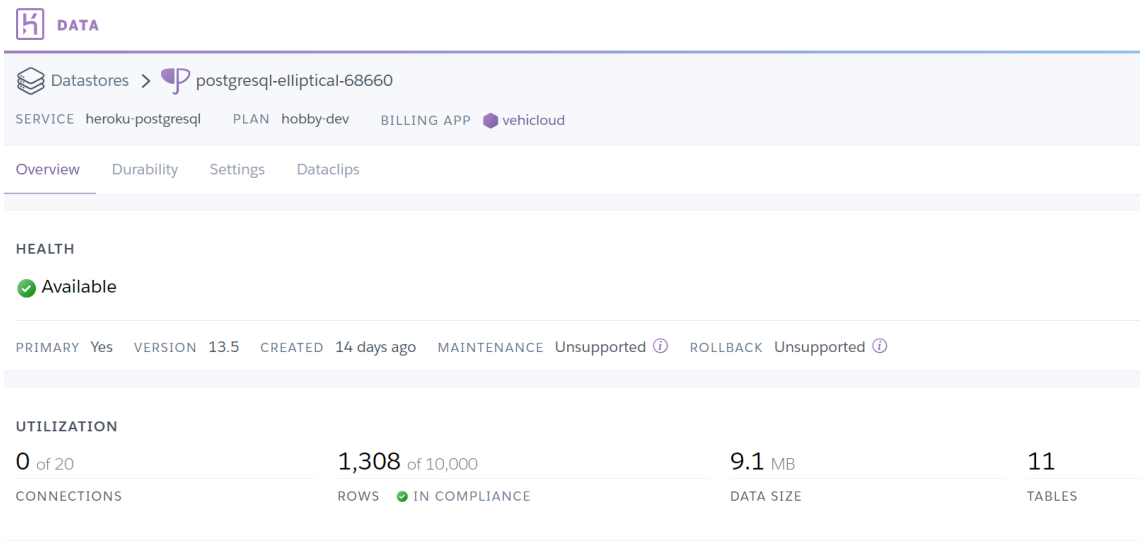


Figure 24: Information on our PostgreSQL database linked to Heroku

To access the Postgre database, we connect to it using the information on Heroku (host, database, username, password). We then select the corresponding database, and once started, we can query the database in order to only show the data of interest. For example, we chose to use the following query to show only the data retrieved during the last 24 hours. The result of the query will be what is used in Leaflet to show the data on the map, as there is no need to analyse older data that is no longer relevant.

3.4 Data presentation - part1 : Node-RED

The aim being to present the data in the form of a map, that the user can use to have access to the indicators provided by our solution, we have first used Node-RED in order to generate that map. In that purpose we used the web-worldmap node paired with the ui-heatmap node. Unfortunately that first prototype of the map could not generate a heat map the way we wanted to. As a matter of fact, the Node-RED module we used was not compatible with the world map module. In the sense that heat map module only takes a matrix of magnitudes as input and not coordinates. Therefore the heat map could only be displayed on a still image as background but not on an interactive web map.

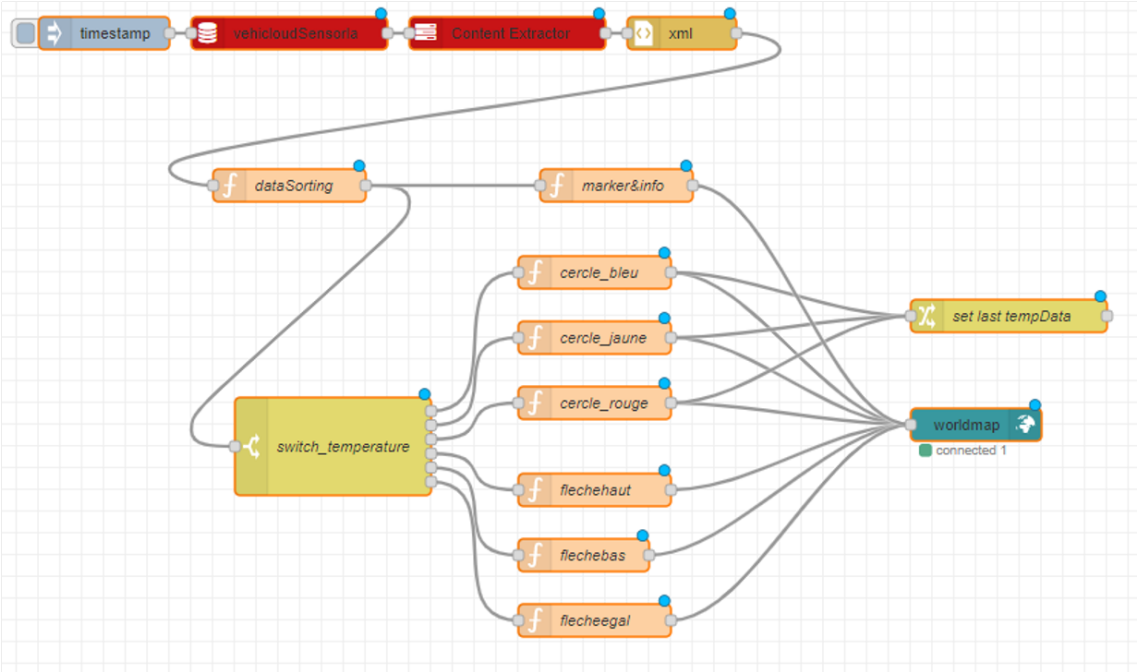


Figure 25: Flow of the first Node-RED Prototype

We still managed to display data on Node-RED. The solution we then settled for at that point was to create a colored circle around the location pinpoint, and fixing different levels for the color to change (in the prototype we arbitrarily chose to show a red circle when the temperature is above 20 degrees, a blue one if the temperature drops below 10 degrees, and a yellow circle if it is between the two values). An arrow over this circle indicates if the value has increased (the arrow points up) or decreased (the arrow points down).

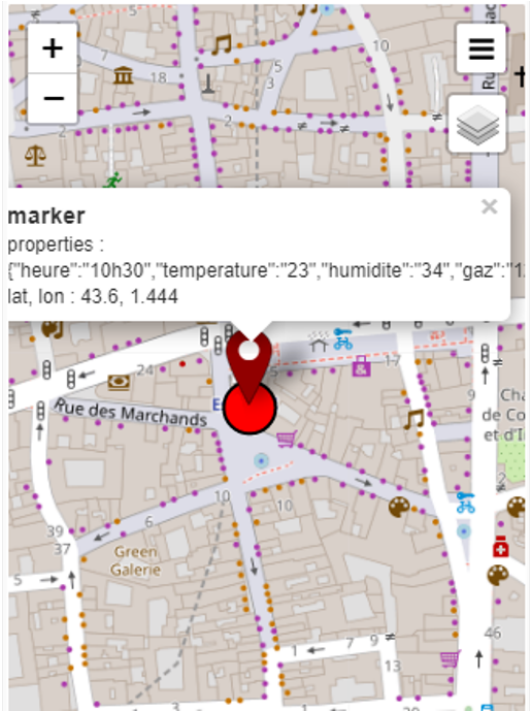


Figure 26: First map prototype with Node-RED

In order to present the data as a map, and to have more control over the way each piece of data will appear on this map, we decided to use Leaflet, an open-source JavaScript library for interactive maps. Leaflet allows to present more easily the data in the form of a heat map and with a better result thanks to the Leaflet-heat plugin, which made it thus our final solution to present the data.

3.5 Data presentation - Part 2 : Leaflet

To implement our solution to represent the data in the form of a map, our final version uses Leaflet, a JavaScript Library that allows us to show the data in the form of a map. The advantage is that it offers a better final render than the Node-RED solution and offers a lot flexibility, allowing us to create layers for the different types of data, which seems like the best solution, in order not to pollute the map with different types of indicator and make the data confusing.

In order to display the map, we need to import the Leaflet JavaScript library paired with the heatmap plugin for the creation of the interactive heat map, and the jquery library in order to be able to import the JSON file containing the data.

```
<script src="leaflet-heat.js"></script>
<script src="jquery-3.6.0.min.js"></script>
```

Figure 27: Import of Leaflet and jquery libraries

This JSON file is imported in the html file creating the map with a jquery function that allows us to import data from a file.

```
$(document).ready(function(){
    $.getJSON("data_15janv_1.json", function(data){
        values=data.values;
    })
    .done(function() {
```

Figure 28: Import of the JSON file containing the data with a jquery function

Once the import is done, we create a table for each indicator (Temperature, Humidity and the two Gases) and a table for the markers. The indicator tabs contain the coordinates and the value of the indicator only. The markers layer contains the coordinates and the value of each indicator, plus the timestamp in order to be able to show at what time has the measure been taken.

```
var markers=L.layerGroup();
var valuestemp=[];
var tabtemp=[];
var tabhumidity=[];
var tabgaz1=[];
var tabgaz2=[];
var tabmark=[];
```

Figure 29: Creation of the tabs containing the data regarding each indicator

We then create the underlayer, which will be the map and set the correct view in order to make it be displayed directly in the right geographic zone (in our case Toulouse city center).

```
var map = L.map('map').setView([43.60742764039704, 1.4449328184127805], 17);

var tiles = L.tileLayer('https://api.mapbox.com/styles/v1/{id}/tiles/{z}/{x}/{y}?access_token=pk.eyJ1IjoibWFwYm94IiwiYS
    maxZoom: 22,
    attribution: 'Map data &copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors, ' +
        'Imagery © <a href="https://www.mapbox.com/">Mapbox</a>',
    id: 'mapbox/streets-v11',
    tileSize: 512,
    zoomOffset: -1
}).addTo(map);

var heatMapPoints = [];
```

Figure 30: Creation of the map underlayer and view setting

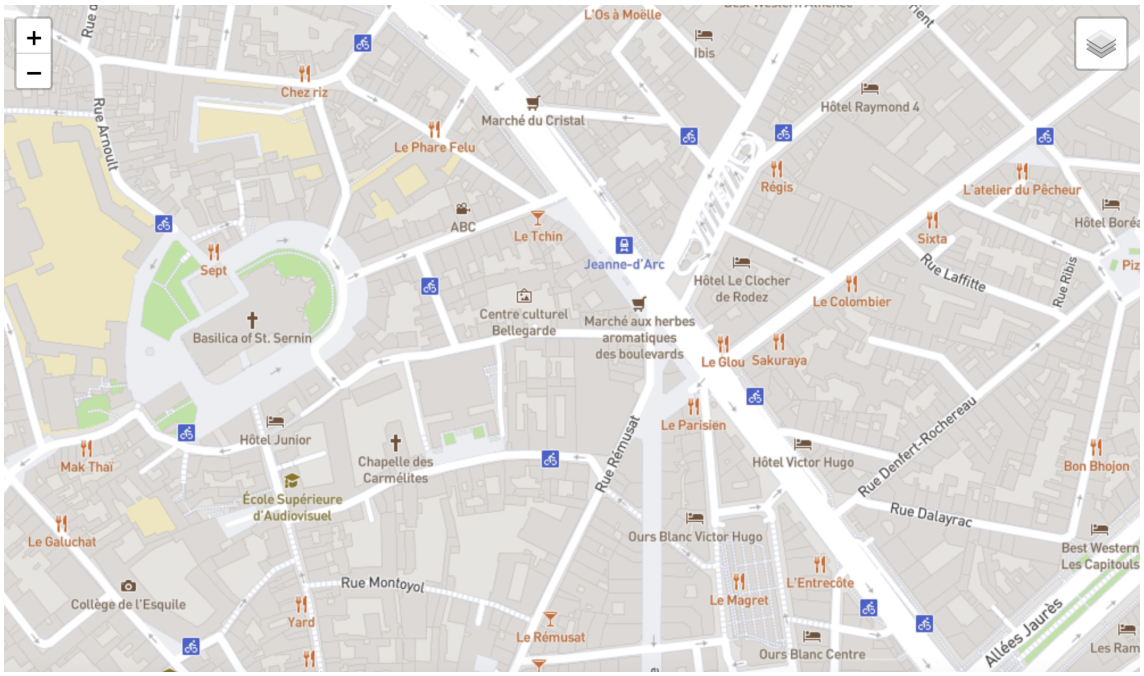


Figure 31: Map view on opening (centered on Toulouse city center)

We can then create one layer per indicator (see figure 32), which will be added on the map in the end.

```
var humiditylayer = L.heatLayer(tabhumidity, {
    radius: 40,
    minOpacity : 0.3,
    blur : 50,
    maxIntensity : 30,
    gradient : {0.3: 'blue', 0.55: 'lime', 0.8: 'red'}
}).addTo(map);
```

Figure 32: Creation of the humidity layer, with the tab created before from the JSON file with the data about humidity (coordinates and humidity value)

The last layer is the markers layers, which at first we did not include at first, but ended up adding because it did not seem very interesting to only have heat maps, which are good visual indicators but don't allow the user to have the precise data value. The marker layer show markers on every point where data has been recorder and on which we can visualize the value for each indicator :

```
//Remplissage markers
for (const element of tabmark){
    L.marker([element[0],element[1]]).bindPopup('<br><b>Timestamp :</b> '+element[6] + "</br>"+'<br><b>Temperature :</b> '
}
```

Figure 33: Creation of the markers layers

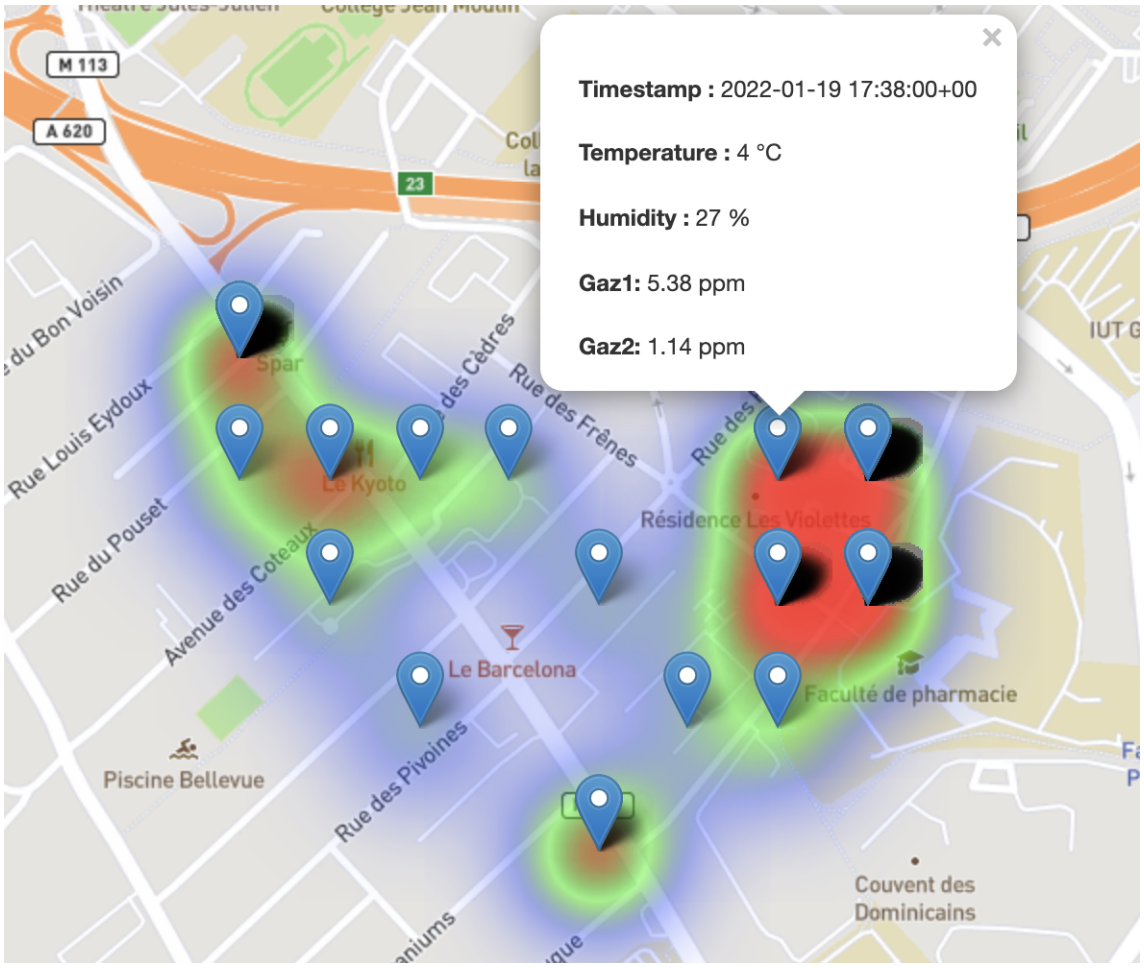


Figure 34: Data visualization when clicking on a marker

We can now finally add all the layers to the map, by indicating the name of each layer which will be displayed, and the underlayer, which is in this case is only the map, but we could provide other underlayers (satellite map views for example);

```
//Gestion layers
var baseLayers = {
    "Mainmap": tiles
};

var overlays={
    "Heat":heatlayer,
    "Humidity": humiditylayer,
    "Gaz 1": gaz1layer,
    "Gaz 2": gaz2layer,
    "Markers":markers
};

L.control.layers(baseLayers, overlays).addTo(map);
```

Figure 35: Layer Management

In the end, we can visualize all these element by opening the html file with a browser, and thus see the map, and select/unselect the different layers in order to only show the data of interest.

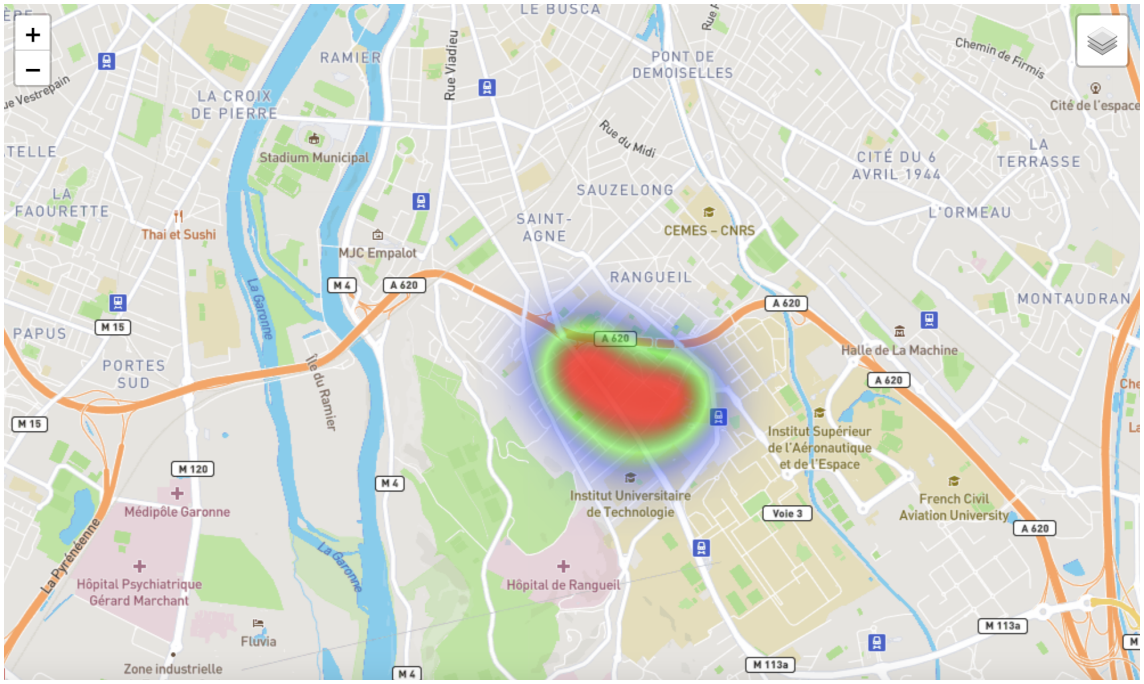


Figure 36: Leaflet map visualization

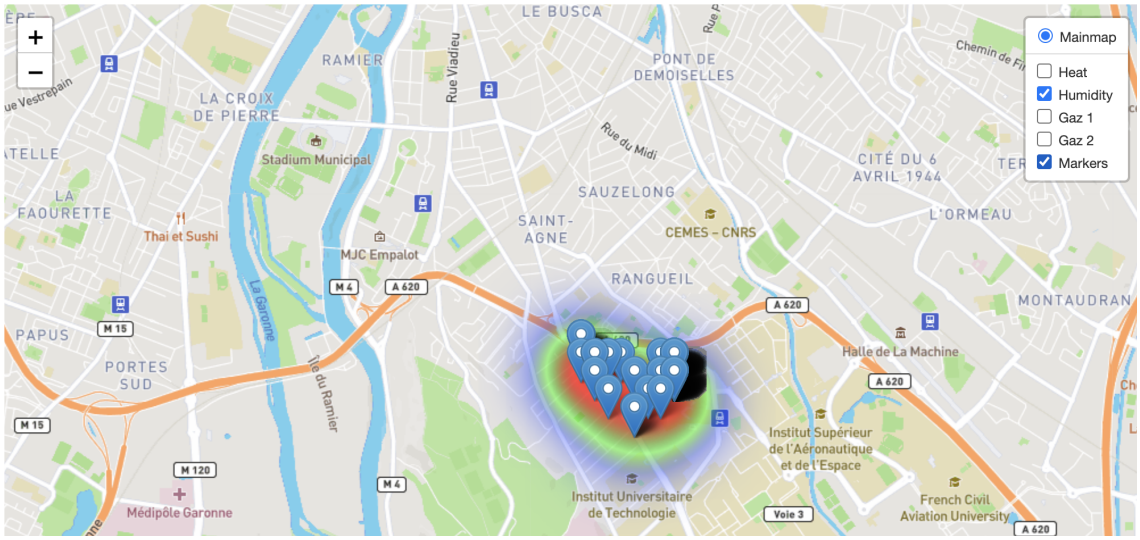


Figure 37: Map layers

3.6 Data retrieval from the API

To generate the JSON file containing the data directly from the API with the most recent data, we used Heroku DataClips, which allow us to query the Heroku Postgres Database and share the results. The only query we ended up running was the very simple query that select.s the most recent data from the database, here, the data from the last hour. It is of course possible to increase or decrease this time interval according to the data we want the map to show. The result of this DataClip can be shown directly on the browser in the form of a table or be downloaded as a JSON or CSV file. In our case, we will be interested in the JSON file

```
1   select *from vehicloud_sensordata
2   where time >= Now()- '1 Hour'::INTERVAL
```

Figure 38: Dataclip query

3.7 Future interface improvements

Our interface is far from perfect in order to adequately achieve our goals. The first flaw being that all our files are stored locally. We first download the JSON file from the Heroku dataclip query. These two steps should be automatized in order not to require any manual handling to run the query and download the JSON file, which is actually the case. Then, the JSON file and the html file should no longer be stored locally, they should be hosted online. Finally, in order to provide the user with a friendly and more adequate interface, we have come to the conclusion that since our goal is to provide the users with instant data, the best format would be an application, accessible on a phone, which would be more adequate for daily use, and more user-friendly since nowadays applications have become way more popular than regular browser interfaces, less adapted to our purpose.

4 Casing and integration in the environment

Here we describe the process of integration of our embedded system on a bike in a real case scenario. This includes the design of a casing for the sensors, ESP board and battery to protect the electronics from the outside elements, and the fabrication of this casing using 3D printing.

4.1 Environmental constraints

There is an inherent contradiction in this context of integration, the system must be protected from its environment but it also must be exposed to ambient air in order to collect relevant data. The subsequent solution has to allow a circulation of air in the casing in order to expose the sensors to the ambient air while blocking any liquid from entering it and creating short circuit in the electronics.

On a bike, the system can also be exposed to a lot of vibrations and other mechanical forces and shocks that can damage components. To prevent this kind of damage, every component must be properly attached and secured in the casing. For that matter, all three sensors are either bolted or screwed into the casing walls. The ESP32 board doesn't have any holes to pass screws through like the sensors do. Therefore, to keep it in place we had to design a specific slot that allows it to be easily attached to the casing and guarantees access to the two buttons at the back of the board while the board is closed.

Finally, the GPS sensor is linked to an external transmitter which has to be outside of the case in order to transmit in ideal conditions. We also need an access for power, from the power source to the ESP board. Both of these access ports represent a risk of exposure to humidity.

4.2 Casing conception

For the conception of the casing we chose to use Fusion 360 from the Autodesk suite for several reasons. First, Fusion 360 is more recent than some more conventional CAD programs (Computer-Aided Design) like Catia or Solidworks, which means it user interface is more intuitive and user friendly. Moreover, those commonly-used programs require an expensive licence to be used in full capacity whereas Autodesk proposes a free licence for non company projects. Finally, Fusion 360 is, by design, optimised for 3D printing which allowed us to quickly prototype and implement our system.

To address the problem of exposure to ambient air for the gas and air sensors without exposing the rest of the hardware to the risk of ambient humidity, we decided to divide the casing in two compartments. One that allows an air flow to pass through the temperature sensor and the gas sensors, and a second compartment isolated from the environment as much as possible with a single opening through the first compartment to allow the wire connections between sensors and the ESP board. This opening can later be sealed once all components are in place and connected; in our first prototypes this is simply done with hot glue.

To prevent vibrations and shocks from damaging the electronics, every component is screwed into place thanks to specific holes already included in the sensor boards used in our prototype.

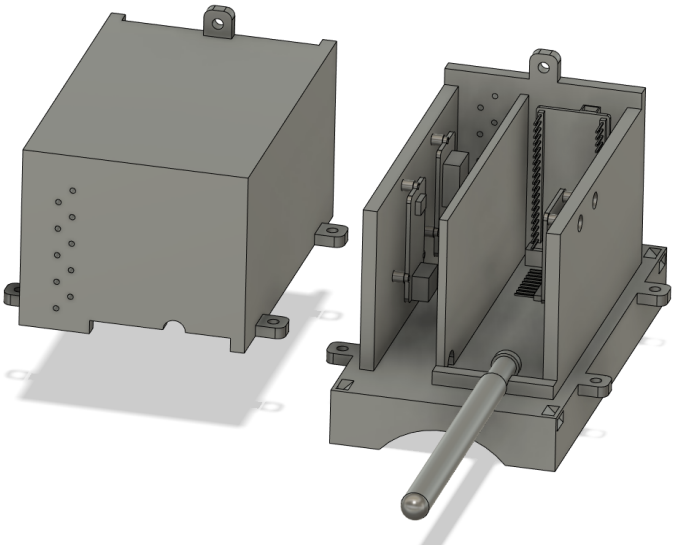


Figure 39: CAD design of the casing for the integration process

The prototype presented in this stage of development does not address the insulation issue. In addition, for easier and faster prototyping we chose not to lock in place every component and maintain easy access to the electronics. For the same purpose the battery is not included in the casing in this version. We intend to correct these issues in a later version of the casing design, once the implementation is completely done.

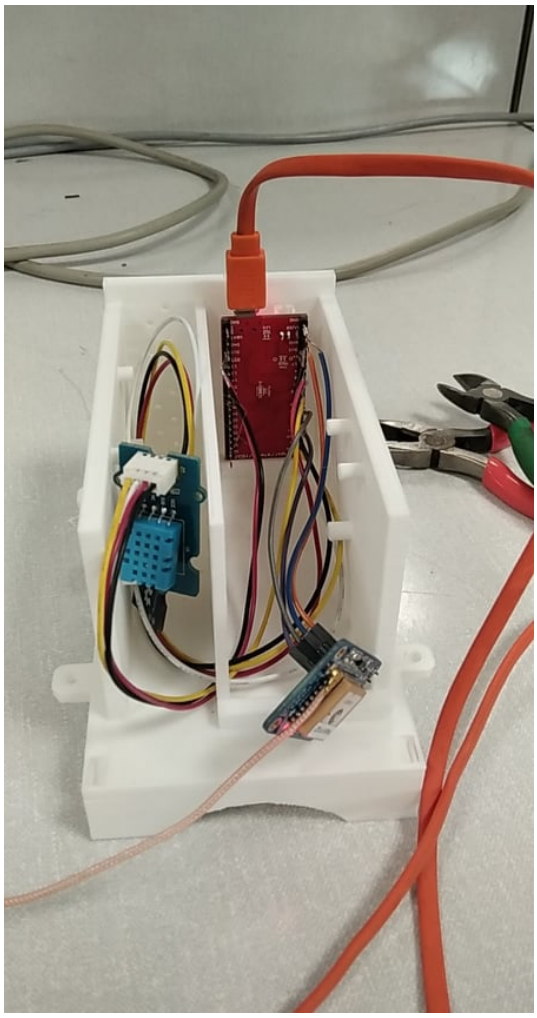


Figure 40: Picture of the embedded system during the testing period

4.3 3D printing

Once the case was designed we needed to prepare it for 3D printing. The first step is to import the STL file into the Utlimaker CURA software.

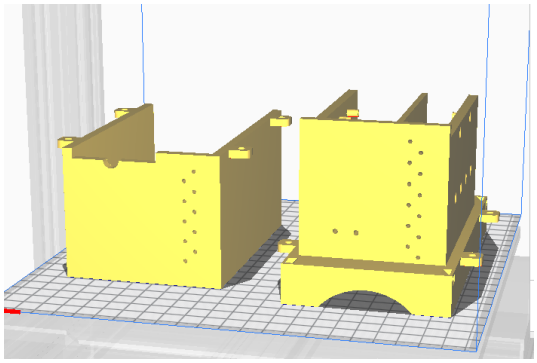


Figure 41: STL file imported in Cura

We chose the Wanaho D12 230 3D printer and can now adjust the properties. As we wanted to save time, we printed the object in low quality, that is to say a higher layer height, less infill density, and less strong infill pattern. The differences between the ideal settings with which we would have liked to print with and the parameters used are presented in a table below.

print settings	ideal value	used value
layer height	0.2mm	0.28mm
wall thickness	0.8mm	0.8mm
top/bottom layer	1mm	1mm
speed	60mm/s	60mm/s
print temperature	195°C	200°C
plate temperature	60°C	60°C
infill pattern	grid	Cubic Subdivision
infill density	20%	10%
generate support	yes	yes
support overhang angle	70°	50°
support pattern	ZigZag	ZigZag
estimated printing time	25h43min	16h22min

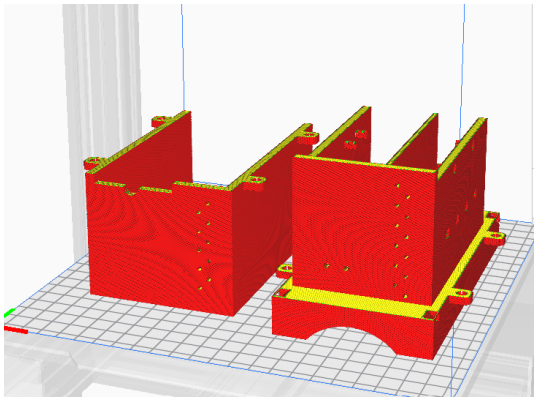


Figure 42: Case after slicing

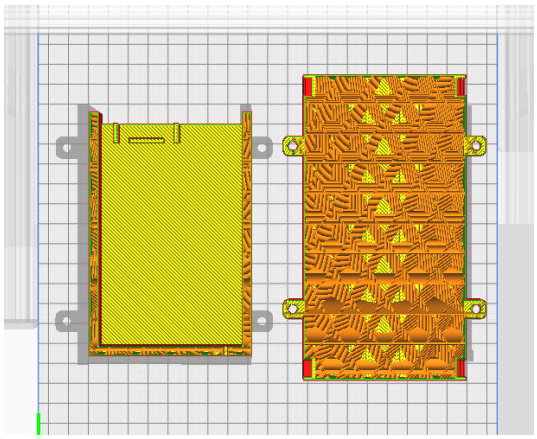


Figure 43: Cubic subdivision

With a view to fast prototype, these settings permitted us to save almost 9 hours of printing, but the casing has some defects.

The supports were quite difficult to remove and it slightly damaged the bottom of the case. The low infill density, makes the prototype fragile. It makes it very light but it could not resist to impacts or difficult weather conditions. Despite this, our solution is satisfactory to make the first tests and prove our concept, we could embed our system on a bike and simulate a first use in real conditions.

5 Proposition for the deployment of the system

In this part we will focus on the key points to improve in our prototype in order that it could be deployed in the city of Toulouse.

5.1 Energy matters regarding WSN

In every WSN for IoT, the energy issue is important since the sensors do not have an unlimited source of power supply. In our case, even if we managed to save energy when we could (GPS decreased mode, BLE connection only when stopped) we approximated the energy consumption of our system between 40 and 60 mA depending on the modes. This is why, in order to have the most autonomous sensor possible we thought about using the energy of the bike thanks to a dynamo. Indeed, by measuring the percentage of battery tanks to a fuel gauge (electrical component measuring battery state) we could enter in a decreased mode of our firmware when our battery is low and wait for the dynamo to fill our battery again. A battery of 5000 mAh would take about 6 hours of riding at 15km/h to be fully filled again, see the figure bellow [?] comparing dynamo of 5V output.

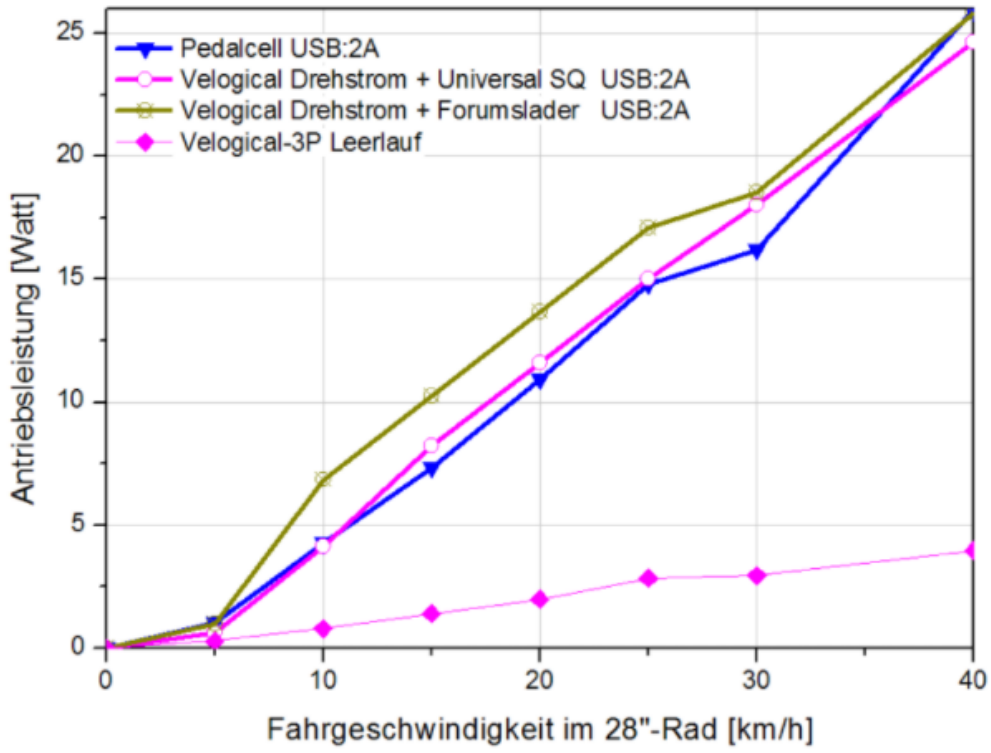


Figure 44: Dynamo power efficiency

5.2 Improve security when connecting to a bike

For the moment, every device called VehicloudXXXX can connect to the gateway. That is not very secure and anybody can easily name a device with an appropriate name and connect to the platform. Which means that we should also pay attention to security. A first idea is to verify that the MAC address of the device matches the MAC address of a bike on our network. In order to do

that, we would have to stock all the bikes' MAC addresses into a database in a REST API and, when we extract a MAC address, compare it with the database to find a corresponding one. If so, that means the device we try to connect indeed belongs to our bike network.

5.3 Improve the backend security

As of this first version of our product, serving as a proof of concept, anybody can post erroneous data to our database through the API, as long as the data format respects the format of data required in each field. It is entirely possible as of right now to flood the database with false data. This issue could be tackled by setting up proper authentication for data inflow, as well as adding some filtering methods to this incoming data (for example, delete any dataframe where the temperature is above 50°C). This would make our application more robust to outside influence. Other security issues exist within our data pipeline, issues that definitely need to be addressed if this project ever evolves from the proof of concept stage.

5.4 Benefits and estimation of the cost

As we explained previously, the tool we are developing is dedicated to municipalities and decision makers in order that they can the best decision in term of efficiency to c. Institutions are always worried about the cost of such a solution. The major benefit they have is to be able to justify their action thanks to accurate data. But our solution also can provide a huge coverage of the city with only 1000 bikes and 150 gateway which represent around half of the actual bike network in Toulouse. We estimated the cost of our equipment for a bike to 50 euros, in total this would cost around 60,000 euros in hardware. Without pricing the installation and maintenance, it is an attractive price for a big city like Toulouse.

Conclusion

In this paper, we have explained the different technical aspects of Vehicloud, our IoT proof of concept product for monitoring cities. From it's inception, this project had the goal of visualizing environmental phenomena and pollution at the finest level of a city. With our proof of concept, comprised of both a hardware (device and gateway), and software (API, database and mapping solution), we have attained this goal. We managed to embed different sensors on a bike and show on a map every area we passed through. Many improvements need to be made at different levels, such as hardware iterations for the casing and sensors, as well as software updates to robustify our backend. Despite this, this project has as a result a full product, that span the creation of data, all the way to it's processing at the front end, in order to present a clean result to the user.

References

[1] G. Shaddick, M. L. Thomas, A. Green, M. Brauer, A. Donkelaar, R. Burnett, H. H. Chang, A. Cohen, R. V. Dingenen, C. Dora, S. Gumy, Y. Liu, R. Martin, L. A. Waller, J. West, J. V. Zidek, and A. Prüss-Ustün, "Data integration model for air quality: a hierarchical approach to the global estimation of exposures to ambient air pollution," vol. 67, no. 1, pp. 231–253. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1111/rssc.12227

[2] Toma, Alexandru, Popa, and Zamfiroiu, "IoT solution for smart cities' pollution monitoring and the security challenges," vol. 19, no. 15, p. 3401. [Online]. Available: https://www.mdpi.com/1424-8220/19/15/3401

[3] S. Kaivonen and E. C.-H. Ngai, "Real-time air pollution monitoring with sensors on city bus," vol. 6, no. 1, pp. 23–30. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2352864818302475

[4] F. Corno, T. Montanaro, C. Migliore, and P. Castrogiovanni, "SmartBike: an IoT crowd sensing platform for monitoring city air pollution," vol. 7, no. 6, p. 3602. [Online]. Available: http://ijece.iaescore.com/index.php/IJECE/article/view/8729

[5] C. Bertero, J.-F. Léon, G. Trédan, M. Roy, and A. Armengaud, "Urban-scale NO2 prediction with sensors aboard bicycles: A comparison of statistical methods using synthetic observations," vol. 11, no. 9, p. 1014. [Online]. Available: https://www.mdpi.com/2073-4433/11/9/1014