Report

Javier CLADELLAS

August 2022

Contents

List of Figures

List of Tables

Introduction

1 Context

In 2021, the residential and tertiary sectors represent the 47% of the total energetic consumption of France, equivalent to 761.9 TWh consumed. [1]

As these numbers have been increasing during the past years, working on building's psychrometrics becomes more and more important. It can help on reducing the ecologic impact, and on improving the thermal comfort and productivity of the building's occupants.

In this context, the *Ktirio project* focuses on developing an online platform of services for energetic simulation of buildings. It embeds a bench of tools ranging from automatic model generation to their simulation, including comfort and energy efficiency assessment, defect detection, design optimization and optimal control. Ktirio services are based on physical models (heat and moisture transfer, fluid dynamics, ...) and advanced mathematical and numerical methods such as data assimilation that incorporated sensor data.

2 Previous Work

The work done in this internship is an extension of the work done during the Master's project. Previously, a dashboard providing monitoring services containing real time indicators based on measured data was developed. We can view what the dashboard used to look like before the internship in the figure 1.

At this state, the data visualization platform counted with the following features:

- Requesting the historic values of temperature, humidity and Open/Close sensors (provided by *Cisco Meraki*) located at the Synapse Concept's building and an office of the University of Strasbourg.
- Plotting the historic data and making it interactive.
- Providing frequency histograms.
- Computing correlations between all the different sensors.
- Providing multiple customizable comfort indicators.
- Importing external weather data.

Figure 1: State of the dashboard at the begining of the internship

- Grouping the sensors by thermal zones or spaces using an IFC file.

Complete information on the Project's work can be found here.

3 Objectives

The main objective of the internship is to add features and enhance the previously developed tool. This will facilitate the understanding of energetic models.

The planned enhancements can be categorized in the following categories:

- New and complete information.
- Performance optimization and deployment.
- Computation of new indices and statistical data.
- 3D visualization tools.
- Customization and flexibility.

Concerning the performance optimization, a local database is planned on been implemented. By doing this, the time taken for requesting the sensor's data is going to be reduced significantly and the dashboard will take less time to load. Also, the code is going to be optimized as possible to reduce computation time.

The dashboard is going to be deployed to be accessible by anyone when a stable version of the code is reached. Subsequently, a user authentication system is goin to be added.

This will allow new companies that want to use the visualization service to privately access their data.

New sensor data providers are going to be added to the dashboard to provide more information on the building, depending on the type of data collected by the sensors. For example, air quality and pressure sensors will be available. Also, missing historic data is going to be filled using machine learning algorithms on the time series to have a complete dataset. Predictions of the future values will be available for multiple time windows.

As for the statistical data, the uncertainty of the available indicators will be assessed using Monte Carlo simulations. Environmental and containment indicators will be implemented.

A tool for visualizing the building and its sensors is going to be added. This will allow to visually select the sensors of interest and access more information on the building, making the dashboard more interactive and user friendly.

Finally, the dashboard's code is going to become authomated and will reduce the quantity of coding needed to be done by the administrator. The authomation will facilitate adding new data providers and new sensor models. The data shown in the interface will be customizable by handling a configuration file.

Chapter 1

Database construction

As the data shown in the dashboard is currently directly requested from the data providers, it takes a significant amount of time to query the information. In order to reduce the time for requests, a local database is planned on being implemented holding all the data needed for the dashboard to work.

New information will be recurrently inserted in the database, instead of being requested each time the dashboard loads. The requested data will be treated in real time and stored once processed. These asynchronous requests will be done by the server, each day, in a given time of the day.

The figure 1.1 shows how requesting data is handled without the feature, and the figure 1.2 show the dataflow with the feature implemented.

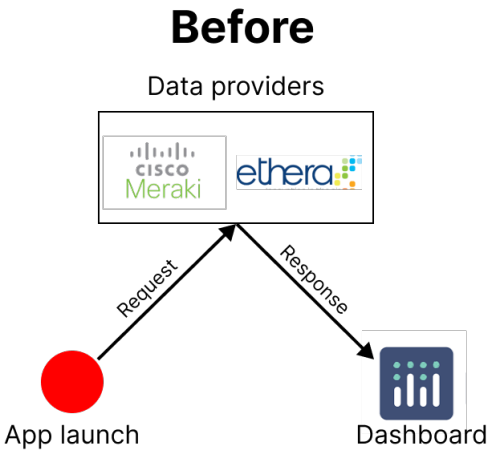


Figure 1.1: Dataflow without the use of a local database.

Implementing this feature will not only improve the dashboard's performance by speeding up the request times, but it will allow plotting bigger amounts of data (longer periods), and will eliminate the risk of losing old information.

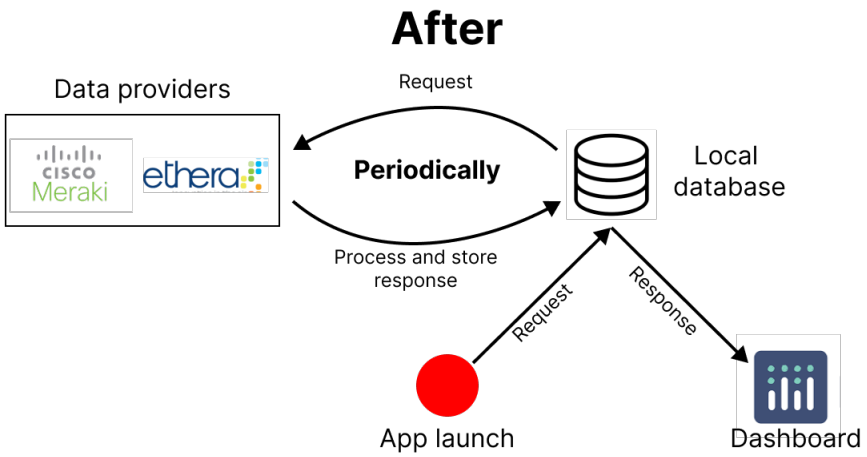


Figure 1.2: Dataflow with a local database implemented.

These enhancements are going to improve the dashboard's user experience and boost some of the models results.

1 Choosing the optimal database

1.1 Data Structure

We currently are working with two data providers: Cisco Meraki and Ethera. The Meraki platform provides sensors for temperature, humidity and Open/Close. Ethera's sensors are for air quality (concentrations of multiple gases), pressure, temperature and humidity.

We must note that we can also get the battery level for each one of the sensors (at multiple timestamps). This information could be of some use later on.

To correctly select a data structure that fits the purposes, we first need to study the structure of the json responses given by the platforms. The requests are performed using the HTTP GET protocol.

For the Meraki API, requesting the sensor's information gives a json response with the following structure:

```
{
    'count': number of sensors (int),
    'sensors':
    [
        {
            'alertProfiles': [],
            'alertingOn': {},
```

```
                        'name': Sensor given name (str),
                        'serial': Sensor serial (str)
                    },
                    ...
                ]
            }

```

When requesting the data for one specific sensor, for a given timespan, the following structure is returned:

```
            [{
                'avg': 23.003906778477752,
                'data':
                [
                    {
                        'ts': '2022-06-06T08:52:00.000000Z',
                        'value': 24.72222137451172
                    },
                    ...
                    {
                        'ts': '2022-06-06T08:54:00.000000Z',
                        'value': 24.72222137451172
                    }
                ],
                'latest':
                {
                  'ts': '2022-06-07T08:44:00.000000Z',
                  'value': 23.55555534362793
                },
                'max': 24.77777862548828,
                'min': 21.66666603088379,
                'serial': 'Q3CA-3MQJ-HE63'
            }]

```

Note that missing values do not appear in the response. Concerning the Open/Close sensors, the structure is conserved, but only values that are equal to 1.0 (Open) are sent.

As for the Ethera sensors, the structure is very similar. We request the values by variable name, by device as it is shown on the diagram of the Ethera API architecture 1.3.

The response of a request to fetch sensor values to the Ethera API looks like the following json:

```
            [{
                "time": 1497194345,
                "value": 1.369,
                "errorCode": 0
            },
            {
                "time": 1497194465,
                "value": 2.329,
                "errorCode": 1
```

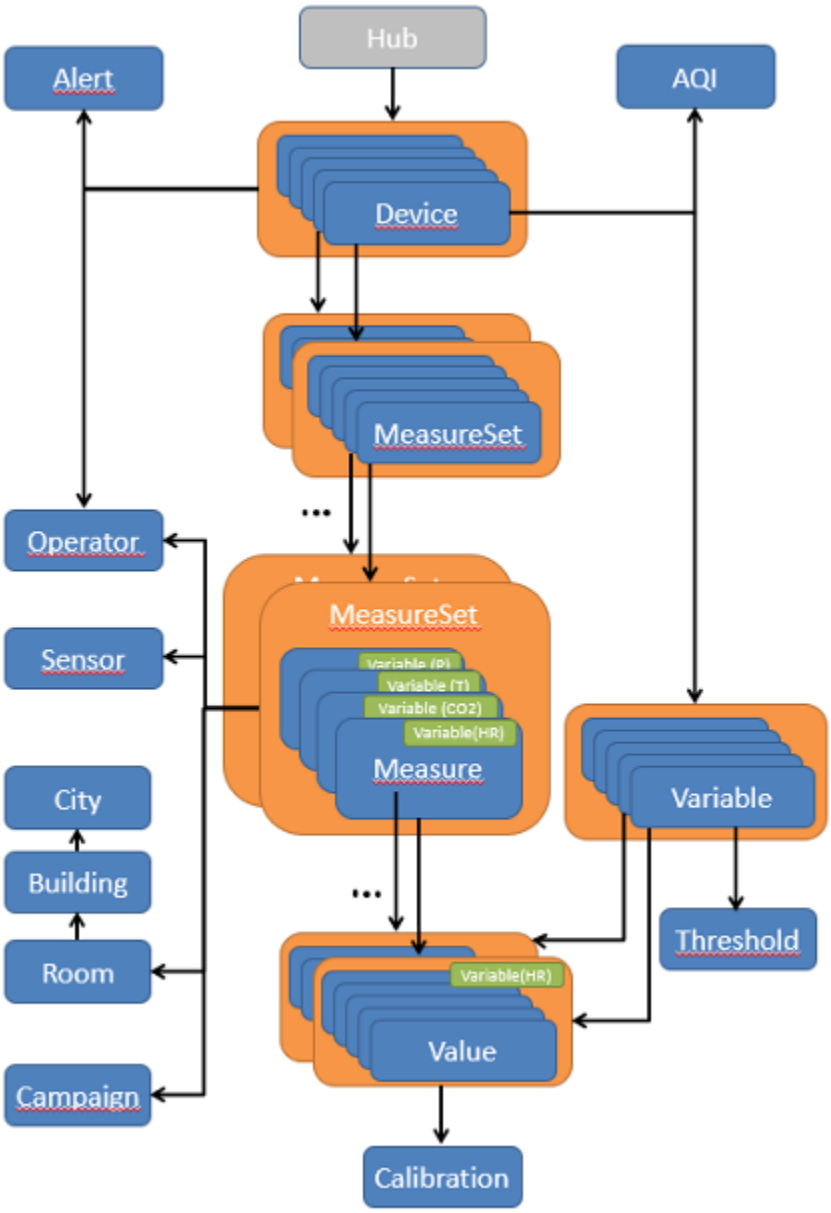


Figure 1.3: Diagram of the architecture of the Ethera API.

```
10                  }]
11
```

Note that the time format is different from the one used by Meraki.

To organize the information and optimize the database requests, we must also understand how the dashboard treats the data in all of the figures shown:

- **Raw data**:
  The functions in charge of plotting raw data take one pandas DataFrame for each

	MT10-temperature-04	...	MT10-temperature-09
2022-06-06 11:00:00	20.48	..	21.65
...	...	...	...
2022-06-07 10:00:00	25.18	...	26.43

Table 1.1: DataFrame used to plot the data of Meraki MT10 temperature sensors.

plot shown. For example, if we decide to show a plot for temperature sensors, a plot for humidity sensors and a plot for the CO2 concentration, the function will take three different DataFrames. These tables are time indexed and contain all the sensors that measure the given metric. The following table shows an example of the dataframe for MT10 temperature sensors:

- **Frequency plots**:
  The same data structure is taken to build frequency histogram plots. The values are then counted. The performance could be improved by directly passing the value counts to the Plotly figures.

- **Indicators**:
  Concerning the comfort indices, one time series for a humidity sensor and one for the temperature sensor is used. In case of thermal groups, the mean of the sensors in the group is taken. The used information consists of the relative humidity, the dry bulb temperature and the radiant mean temperature which is taken equal to the measured temperature. The time series is manipulated row by row using a pandas *.apply* function. Other used data can be a time series containing clothing insulation values, by hour. When selected, a Monte Carlo simulation is done for certain parameters.

  For environmental and containment indicators, three time series are passed to the *AirQuality* class, that calculates the indices. The used time series contain CO2 values, volatile organic compounds values and formaldehyde values.

- **Correlations**
  We first get all available data, calculate the correlations between all the columns (this step is slow and can be avoided), and then depending on the user inputs, columns are shown or hidden. In the case that the user chooses to see the correlations by zone/space, we split the dataframe in four different tables to calculate the weighted average by group, then all of this information is concatenated to the dataFrame containing all the sensors.

We will first explore the possibility of creating a classic MySQL database, the table relations should look like the following figure (fig. 1.4) :

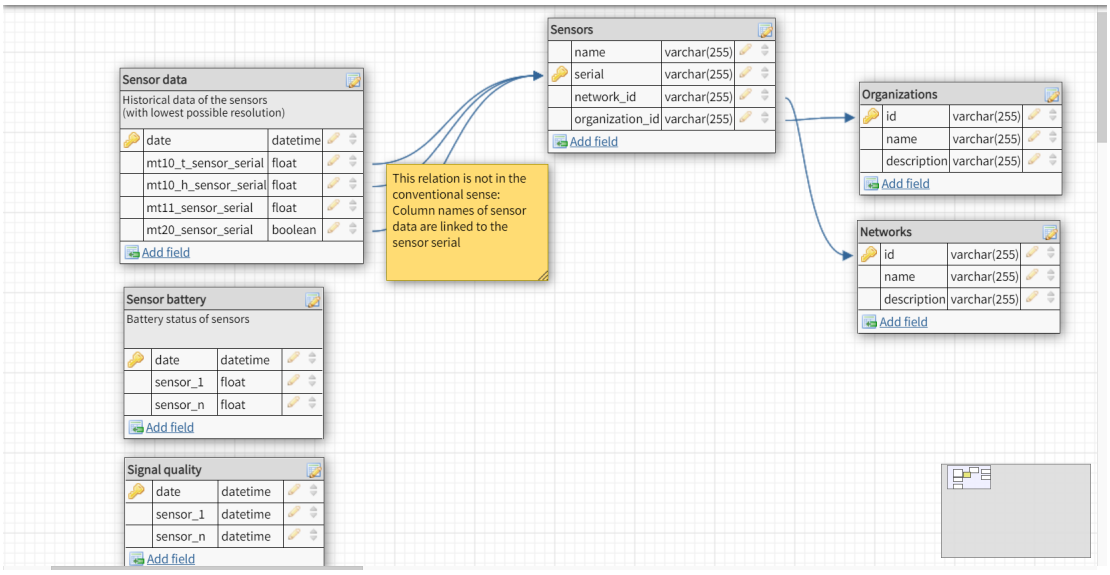


Figure 1.4: Proposed schema for a MySQL database.

1.2 Estimating storage size

We need to make a decision between storage and performance. The data processing can be done each time before loading the dashboard (this will reduce the amount of storage needed) or before storing the data (this will improve the performance of the dashboard, but will need more storage space).

For example, we can reindex the data and fill existing gaps before storing the information. Original requested data could be kept to avoid losing values, but we will need to have a table for each used resolution, which will consume space.

Lets first explore the possibility of just keeping data as requested, without aggregating its values. In this case, lets consider a total of 60 sensors, for a 20 year period and 2 minute resolution. We should have 60*60*24*365*20/120 = 5 265 000 entries in the table. As we have 60 columns, we would have a total of 315 360 000 values. Assuming 40 columns are stored as floats, and 20 are stored as binary data. The float columns would have a size of 210 240 000 * 4 = 840 960 KB. And the binary columns would have a size of 105 120 KB. Giving a total of 946 080 KB, approximatively 946MB.

We can see that the needed storage size is really small. The database could be located almost anywhere.

Now lets consider the case of keeping data as requested (2 minute resolution), but also store the reindexed information. We will use four different resolutions for this: 10 minutes, 1 hour, 4 hours and 1 day. Assuming we have the same number of sensors, the 10 minute table will use 189.2MB, the 1 hour resolution will use 31.5MB, the 4 hour resolution will use 7.8MB and finally, the 1 day resolution table will use 1.3MB. These new tables will hold 229.8MB (24.3% of the original size), giving a total of 1 175.8MB

We can conclude that storing reindexed data is not that harmful to the storage size, as

it will only need around 25% more space to do this.

For preparation purposes, we can assume that the database should need to have a maximum storage of 10GB, which is easy and cheap to get.

2 Reconstruction

We first need to request all previous data in order to have a complete dataset. As the GET requests to the data providers (Meraki and Ethera) are limited by its size, we will create a script that requests the information by batches (by looping through different time sections). The way the script (called *requestDataScript.py*) works is the following:

- We select if we want to request the sensors data (1) or merge existing sensor data files (2).
- Then, input the network of the sensors we want to fetch (e.g. "Ktirio" or "Synapse").
- If option 1 was selected, we must enter the source of the data (e.g. "Meraki" or "Ethera").
- We must login to the selected API.
- For each sensor, its data is requested and saved in a csv file named as the sensor serial.
- Repeat this steps for all the wanted networks and sources.
- Once we fetched all the desired information, we can merge all the sensor data from the same network in a csv file by rerunning the script and selecting option 2 at the start.

Finally, we will have the historic data in multiple csv files located in the *resources/historical/* folder.

Note that the script currently works with .csv files, but can be modified to perform these steps and insert the historical data in another type of database.

The script requests information starting from the current datetime (UTC).

3 Data Processing

After recollecting all previous historic data of the sensors, a mechanism that periodically requests new information needs to be added. To start this feature, a script that is planned on being executed by a CRON job by the server is going to be created. A CRON job is a Linux command that can periodically do tasks, in our case, execute a python script. This python file will first read the historic information on the database, then it will request new data until the last date present on the database (for each sensor). Next, the script is going

to process the new incoming data by reindexing it and filling missing values. Finally, new data will be inserted in the DataFrame. Note that the new information will be inserted depending on the resolution of the data (2 minutes, 10 minutes, ...).

Chapter 2

Data imputation

Whenever there is a power cut or an issue that impairs the devices, information is lost for a period of time. It is important to reconstruct the missing period to improve the models performance, and get a better comprehension of other metrics.

In order to imputate missing data, we will implement an algorithm that was created by Yanick Stoll, which uses a random forest.

First, the Random Forest machine learning algorithm will be explained. Then, we will see how this technique was used to fill the missing data gaps. Finally, we present the test results and the accuracy of the model.

1 Random Forest Algorithm

Random forest is a machine learning algorithm that works by combining multiple decision trees. In order to understand this model, we must understand how decision trees work.

The decision tree algorithm splits the data by a certain feature in each tree node, called decision node. The statistical individual then arrives at a final decision, which is represented by a leaf node.

A visual example of a decision tree can be seen in the figure 2.1.

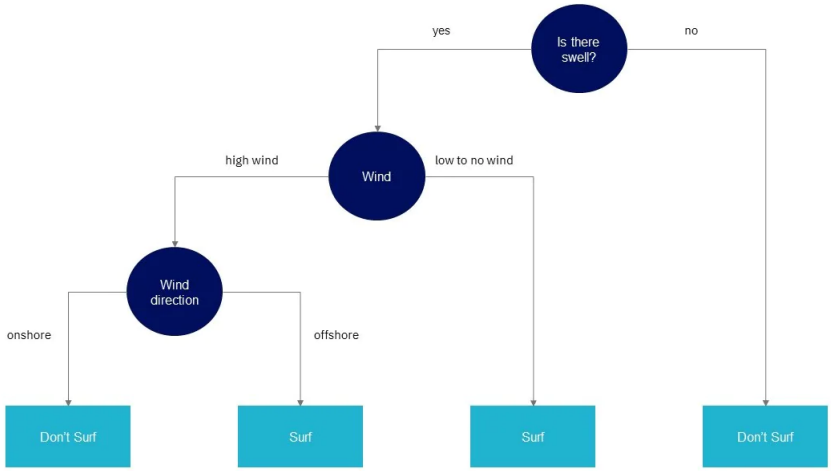


Figure 2.1: Example of the schema of a decision tree, from *ibm.com*.

As said, the random forest model is an ensemble method. It means that the predictions of multiple decision trees are aggregated to select the most popular result, as seen in the figure 2.2. The random forest algorithm takes three hyperparameters: the node size, the number of trees and the number of features sampled [2]. They are really useful algorithms for classification, but also for regression tasks.

By using an ensemble method, we reduce the risk of overfitting the model and we can easily identify which features are most important.

This powerfull tool can also be used to fill gaps of missing data in time series, as it will be seen in the next section.

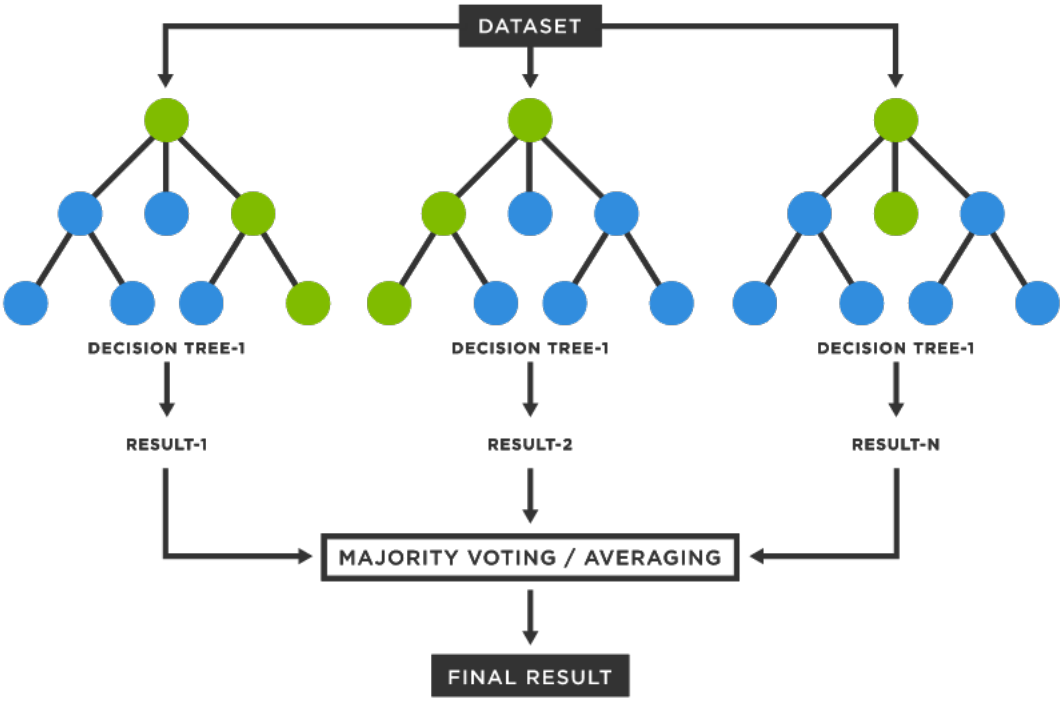


Figure 2.2: Schema of the random forest algorithm.

2 Imputation algorithm

The algorithm written by Yanick Stoll uses a Random Forest regressor to fill the missing information present in time series of the sensor data. The *Impute* class performs the following:

- Identifies gaps in the data.
- Trains a random forest.
- Artificially generates gaps and tests the method.
- Fills the missing data in multiple time series.

The filling algorithm performs a really interesting step in order to improve the models accuracy. It trains itself considering the previous and future data from a gap. We can pass a whole dataframe to the class with columns containing multiple gaps, but only the data from the same column will be considered for predictions. This means that the missing period is reconstructed from each extremity of the gap. The training step can be visuallized in the figure 2.3, and the figure 2.4 shows how the missing data is completed, once the model is trained.

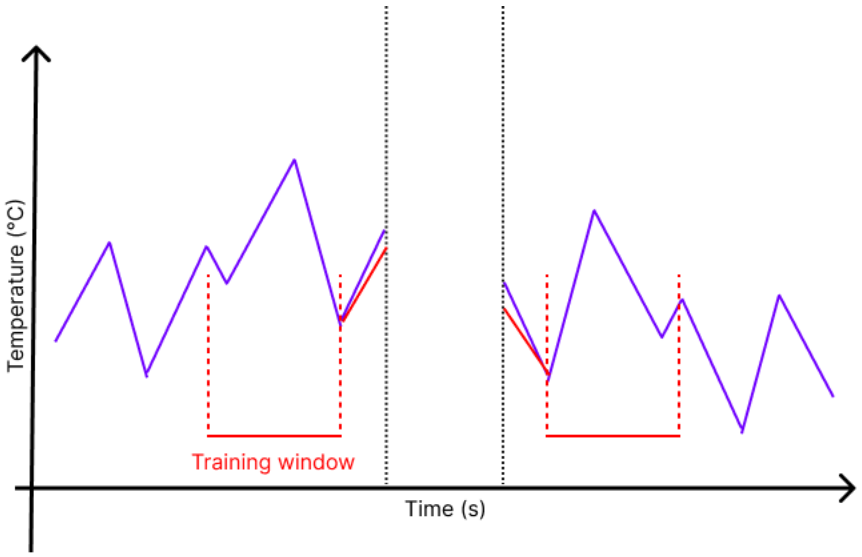


Figure 2.3: Diagram of the training step in the data imputation algorithm.

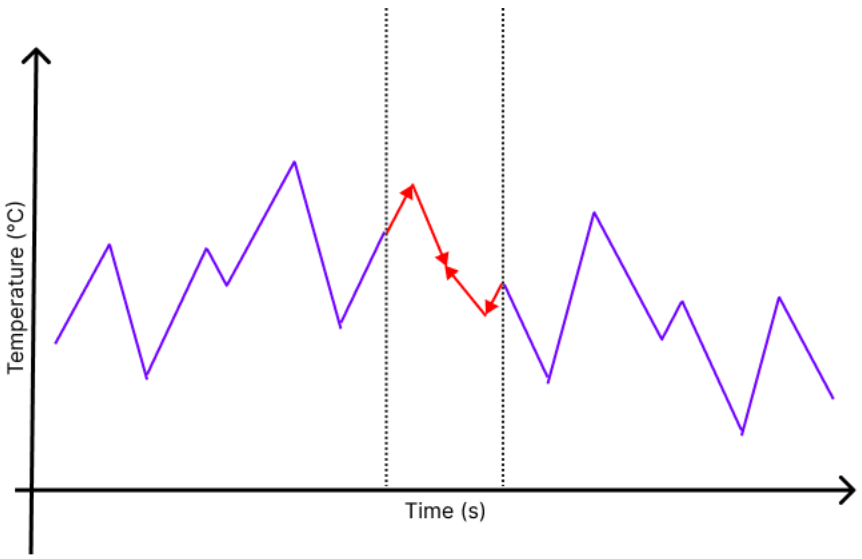


Figure 2.4: Diagram of the the data imputation.

3 Results

Model testing

In order to test the imputation model, we will generate gaps of missing data and then predict the missing values to compare with the true ones. We will test the algorithm for a temperature sensor. Also, gaps of three different lenghts will be tested for each resolution: 1 day, 5 days and 30 days.

We can see an example of a test plotted in the figure 2.5 for a temperature sensor.

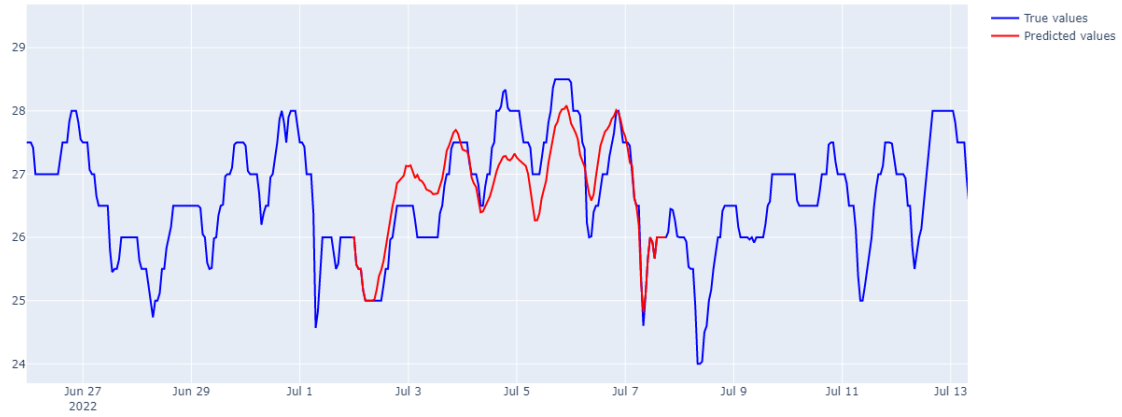


Figure 2.5: Data imputation test for a temperature sensor of the Ktirio Network (Meraki). Missing period = 5 days, resolution = 1 hour.

In the table 2.1 we can compare the mean squared error of the tests for multiple gap lenghts and resolutions. The R-squared metric can be viewed in the table 2.2

MSE	1 day	5 days	30 days
10min	0.02	1.61	-
1H	0.22	0.6	6.7
1D	0.01	0.12	5.8

Table 2.1: MSE by missing period length and data resolution.

From these tables, we can deduce that for small gaps, using a low resolution can lead to better predictions. This is because we give the model more data to train, and certain features become visible. For example, as the values are usually rounded by the sensors, small constant periods appear in the data. These periods are not visible to the algorithms for higher resolutions.

R-squared	1 day	5 days	30 days
10min	0.997	0.8	-
1H	0.97	0.93	0.23
1D	0.99	0.98	0.27

Table 2.2: R-Squared by missing period length and data resolution.

Subsequently, for long periods of missing data, we obtain better results by using a big resolution (a day for example). This is due because the model predicts significantly less values.

It should be noted that using small resolutions on big amounts of information takes a high computational cost and an important amount of time. This is why, unfortunately, a 30 day gap cannot be filled with a 10 minute resolution, as it can take several hours to complete.

Implementation

Once the data imputation algorithm has been tested and adapted, it needs to be implemented on the dashboard and database code to perform the wanted behaviour. First, historical data needs to be imputed. That is, data from when it was first available until the day that the historic data was requested. This step will complete all available information.

Next, data must be imputed before being stored as reindexed in the database. This step will be performed during the processing part of the CRON job. When data is requested from the data providers, it is then reindexed, imputed and stored in different tables.

Finally, we must show in the dashboard which part of the data results from an imputation. To do this, a flag indicating if a value is imputed is added to the data table. By using this flag, the functions in charge of plotting the data were manipulated to show the imputed data as dash lines. This can be seen in the figure 2.6.

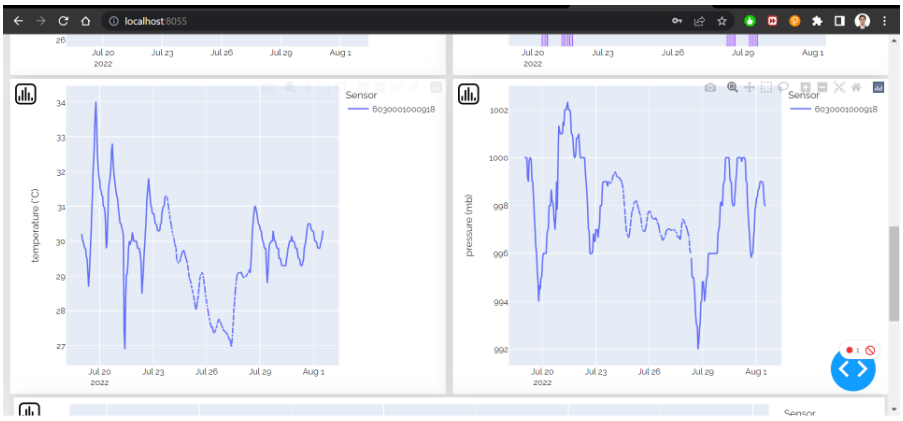


Figure 2.6: Screenshot of imputed data (dashed line) present on the dashboard.

Chapter 3

Dashboard automation

Before the start of this internship, the dashboard was heavily dependent on the data requested. For example, it could only show data from the Meraki sensors and the data provided should exactly be the one from the MT10, MT11 and MT20 sensors.

A problem for this was that if we wanted to add a new sensor model, or request data from a different data source, a big amount of code was needed to be modified.

This section focuses on two improvements: facilitate adding new incoming data and allow customization. All of this with the objective of reducing the amount of coding needed to be done. Even if one is not familiarized with programming, the information shown in the dashboard can be easily customized.

To do this, a configuration file was implemeneted in json format. This file is not meant for the final user to be modified, but by a designated administrator. The file takes care of the following points:

- Create new tabs and chosing which data to show in each tab.
- Choosing which information to request, by including or excluding sensor models or metrics.
- Add new data sources, even fetch data from CSV files.
- Customize the plot type (line plot or bar chart) and define the unit.

1 Configuration file

The configration file is located in the *dashboard/callbacks/config/* folder. It contains three main sections:

```
1    {
2        "sources":{},
3        "data":{},
4        "metrics_info":{}
5    }
6
```

sources

The sources section holds all the needed information for the data providers. The structure is the following:

```
{
    source name 1 : { #Same used in the request callback.
        "org_id":, #Optional field needed for some sources.
        network name 1 : { #Will appear on the network dropdown.
            "network_id" : "", #Needed by the APIs, can be "".
            "alt_name" : "" #Prefix of the data given by the API,
                            it this field is provided it cleans
                            the column names of the dataframes.
        },
        ...,
        network name n :{
            "network_id" : "",
            "alt_name" : ""
        }
    },
    ...,
    source name n:{
        ...
    }
}

```

This section only needs to be modified if we start working with a new sensor that works with a new API, or if we start working with a new database type.

data

The data section holds the information shown in the dashboard. It separates the networks and defines the tabs shown.

```
{
    network name 1 :{
        tab_name_1 : [
            {
                "source":"", # Needs to be in the sources section.
                "sensor_model":"", #Model or serial of the sensor.
                "metrics":[""] #Metrics that the sensor measures.
            },
            ...
        ],
        ...,
        tab_name_n : [
            ...
        ]
    },
    ...,
    network name n : {
```

```
            ...
        }
    }

```

Each tab section holds all the sensors and metrics that we want to show in each tab. The tab names on this file need to be written in *snake_case*, as the underscores will be replaced by spaces and each word will be capitalized.

For example, we can create a tab named "Psychrometrics" that contains all sensors that measure temperature, humidity, pressure and door and windows openings, and another tab named "Air Quality" that holds all metrics related to gas concentrations in the building. We can see this on the following figure (fig. 3.1):

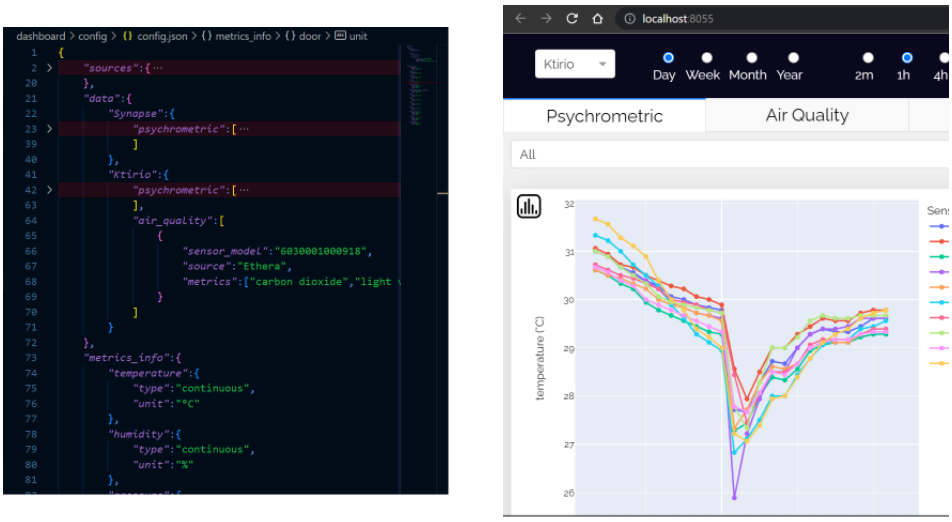


Figure 3.1: Effects of the configuration file on the dashboard tabs.

metrics_info

The last section contains all the used metrics on the dashboard. It defines the data type related to the metric, as the unit used. Note that the metrics used in here need to be present on the column names of the fetched information, otherwise it will not work. The structure is simple:

```
    {
        metric name 1 :{
            "type":"",  # "continuous" or "binary".
                        More types will be implemented in the future.
            "unit":"" #Only used in the plot legends.
        },
        ...,
        metric name n : {
            "type":"",
```

```
            "unit":""
        }
    }

```

Concerning the "type" field, "continuous" means that the data can be represented in line plots. Metrics such as the temperature, the humidity, etc. have this type. Open/-Close sensors are of type "binary". In the near future, a type named "category" will be implemented to work with sensors that provide categorical information, such as buttons.

Comfort configuration file

The configuration json file named *comfort.json* handles some of the parameters of the comfort models. More precisely, in this file we define:

- Which sensor models to consider.
- Information on garments and garment presets for the clothing insulation options.
- A list of physical activities that can be done in the building.
- Clothing insulation extrema values by season.
- Extrema values of comfort parameters for compliance by standard.

2 Dynamically generating figures

When automating the dashboard, one of the biggest challenges was to make the Dash callbacks dynamic. This means that if five sensor models are added in a tab (in the configuration file), five different plots will dynamically appear on the dashboard's tab.

This behaviour is not easily achievable by using the Dash basic callbacks. Luckily, Dash implemented *Pattern-Matching callbacks* in the 1.11.0 version. This feature includes the *MATCH*, *ALL* and *ALLSMALLER* selectors, that "allow you to write callbacks that respond to or update an arbitrary or dynamic number of components". [3] This is why the order of the sensor fields written in the configuration file is important.

In the following example (fig. 3.2) we can see how the dashboard generates all the figures from the Air Quality tab, from one sensor that has multiple metrics.

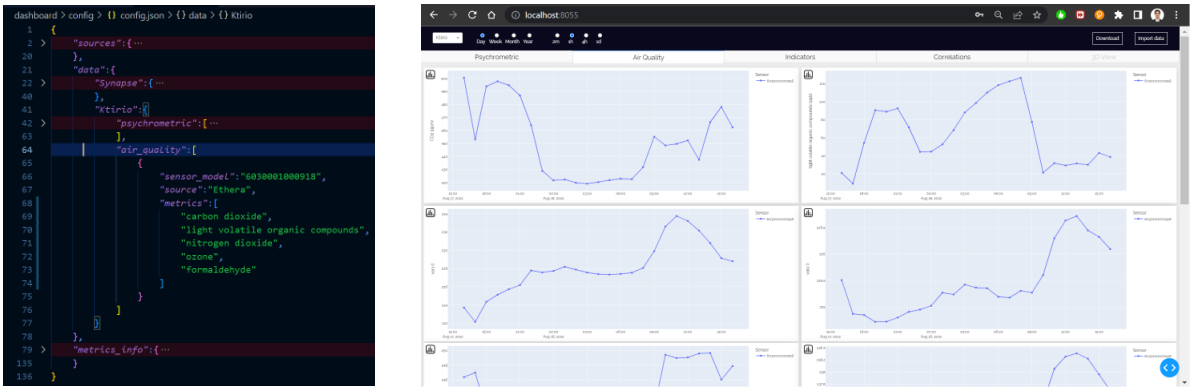


Figure 3.2: Figures generated from the configuration file data.

3 Adding a new data provider

This task was once very hard to achieve, as almost all code files needed to be modified and verified. With the automation of the dashboard, it is now easy to add a new data provider, although some programming is still needed.

To add a new source, these steps need to be followed:

- In the configuration file, add a new source in the "source" section. Specify the networks related to the source and their ID and prefix if necessary.
- Add the wanted sensor models to the desired tab in the "data" section, specifying the source, model and metrics associated. Add a new tab or network if wanted.
- If new metrics are treated, do not forget to add them to the "metrics_info" section, specifying the type ("continuous" or "binary") and the metric's unit.
- In *dashboard/callbacks/requests/* create a new python file named as your new source suffixed by "Query. In this file, you should create a class that handles the requests to fetch the information from the data provider.
- In the *dashboard/callbacks/requests/requestCallbacks.py* file, add a function that fetches the data by network, timespan and data resolution. This function should return a dictionary like the following:
  {*"sensors":dict of sensors names and serials, "data": DataFrame of the data.*}
- Finally, inside the *fetchData* callback, add the following block of code:

```
if YourSourceName in sources:
    resp_tmp = yourFetchingFunction(network,timespan,resolution)
    if resp_tmp["sensors"]is not None and resp_tmp["data"] is not None:
        sensors.update(resp_tmp["sensors"])
        data = pd.concat([data,resp_tmp["data"]],axis=1)
    else:
```

```
7           connection_error_display={"display":"block"}
8
```

Chapter 4

Indoor Environmental Quality

1 Thermal Comfort: Statistical treatment

Previously, thermal comfort indices where added to the dashboard. These indicators where the Predicted Mean Vote (PMV), the Predicted Percentage of Dissatisfied (PPD) and an Energy Savings Chart containing an envelope to check on sensor compliance to the standards.

We remind that the thermal comfort indicators help to find "combinations of indoor thermal environmental factors and personal factors that will produce thermal environmental conditions acceptable to a majority of occupants within the space" [4].

However, even if we posses precise data on temperature and humidity inside a building, we have trouble on correctly calculating parameters that are subjective to the users of the space.

The PMV index takes into consideration the following parameters and returns a value ranging from -3 to 3.

- Dry bulb temperature
- Mean radiant temperature
- External work
- Relative humidity
- Relative air speed
- Metabolic rate
- Clothing insulation

The last three parameters (relative air speed, metabolic rate and clothing insulation) are almost impossible to find their exact value, as they depend on the physical activities the occupants are doing and on the way they are dressed.

This uncertainty can be assessed by the dashboards in two ways: estimating the clothing insulation and physical activities by user input or performing a Monte Carlo simulation on these parameters.

On the dashboard, this choice can be selected by changing the value of the dropdowns in the figure 4.1

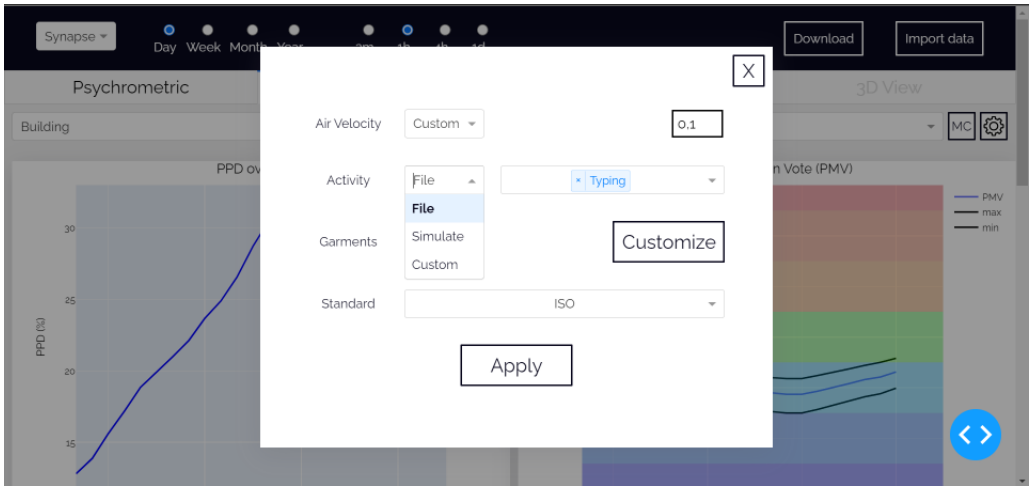


Figure 4.1: Options for comfort parameters.

The "file" option means user input is taken to calculate the comfort indicators. "Simulation" implies that a Monte Carlo simulation is done. Finally, "Custom" lets the user decide which value to assign to the parameters. The three subjective parameters mentionned have these options.

Monte Carlo Simulation

A Monte Carlo simulation consists on taking random values for certain parameters, that follow a given distribution, to simulate the problem multiple times. The average of the simulation results is taken at the end. It is used to estimate the results of an uncertain problem. In this case, it is used to assess the uncertainty of the subjective parameters in comfort indices calculations. It uses randomness to solve deterministic problems. [5]

In relation to the PMV thermal comfort indicator, we will take random values following a uniform distribution. The ranges of the parameters are given in the following table 4.1, according to the ASHRAE 55 and ISO 7730 standards.

As we do not know for certain what probabilistic law the parameters follow, we will start by supposing that they are ruled by a uniform law. When more data becomes available, a better probability function can be applied simulate the parameters.

We can visualize the Monte Carlo simulation on the dashboard by clicking on the **MC** button located at the top of the indicators tab. A modal showing all the simulations for the PMV indicator, as well as the resulting average will appear, as seen on the figure 4.2.

Parameter	ISO 7730	ASHRAE 55
Metabolic Rate	0.8 - 4	1 - 4
Clothing Insulation	0 - 2	0 - 1.5
Relative Air Speed	0 - 1	0 - 2

Table 4.1: Minimum and maximum values for subjective and unkown parameters, by standard.

This is the same plot shown on the indicators tab when the simulation option is chosen for all the parameters. For optimization purposes, the simulation does only 100 iterations.

Figure 4.2: Dashboard modal showing the Monte Carlo Simulation for PMV index.

On the figure on the right, the blue line represents the average of the simulations. The black lines correspond to the 95% confidence interval.

To calculate the confidence interval, we apply the following formula, using the sample of all 100 simulations:

$$[\mu - t\frac{\sigma}{\sqrt{N}}, \mu + t\frac{\sigma}{sqrtN}]$$

Where μ is the mean of the sample, σ its standard deviation and N its size. The parameter t is calculated using the student's t distribution. More specifically, its inverse cumulative distribution function Q_{N-1} with $N-1$ degrees of freedom.

$$t = \left| Q_{N-1}\frac{1-0.95}{2} \right| \tag{4.1}$$

Taking user input

To avoid making simulations, we can gather information of the building's occupants to assess the uncertainty of the subjective parameters. This means, knowing how people are dressed inside the building, and knowing which activities they are doing. Note that this work is still in progress and not fully implemented.

To calculate clothing insulation, the *pythermalcomfort* library provides a dictionnary of garments and presets with their measured values.

In order to approximate the actual average clothing insulation of the occupants, a sensor containing three buttons is planned on being installed at the entrance of the building. The building personel will be asked to choose a button that will represent how cold/warm they are dressed that day. Where they can choose between cold, neutral and hot.

After the occupants have made their choice, it will be averaged and linearly interpolated considering the extrema values for the clothing insulation for each season of the year.

These extrema values are shown in the table 4.2.

Season	min	max
Mid-season	0.4	0.7
Summer	0.2	0.4
Winter	0.7	1.0

Table 4.2: Minimum and maximum clothing insulation by season.

As this button is still not installed, simulations have been made using a uniform distribution. The consequence of simulating this is that we will have equivalent results to the Monte Carlo simulation.

In theory, the sensor will return an integer from 1 to 3: 1 being cold and 3 being hot. Also, these values will be time indexed.

Concerning the metabolic rate parameter, we could ask the occupants to fill a json file indicating which physical activities are practiced inside the building, and how many people are doing it. Then, we take the weighted average of the metabolic rates corresponding to each activity to calculate the PMV. A problem with this, is that the metabolic rate is fixed in time, as each one of the activities is assumed to be stay the same.

As for the relative air speed, the only solution for taking input will be to place one or multiple air speed sensor inside the building. Then, it will be calculated depending on how the metabolic rate is processed.

2 Indoor Air Quality

As we are interested in indoor environmental quality, the air quality plays an important role. Recently, information on the air quality Ethera sensors became available in the dashboard.

To exploit this information, the code capable of computing air quality indicators, written by Jimmy Mousel, was implemented in the dashboard.

Two indicators are plotted: the containment index and the indoor environmental quality. The first indicator considers the carbon dioxide, light volatile organic compounds and formaldehyde concentrations. The second one takes in addition the thermal comfort, sound level and daylight.

These indices use the predicted percentage of dissatisfaction to return a value ranging from 1 to 6, where 1 means very good quality and 6 represents extremely poor air quality.

An example can be found on the figure 4.3

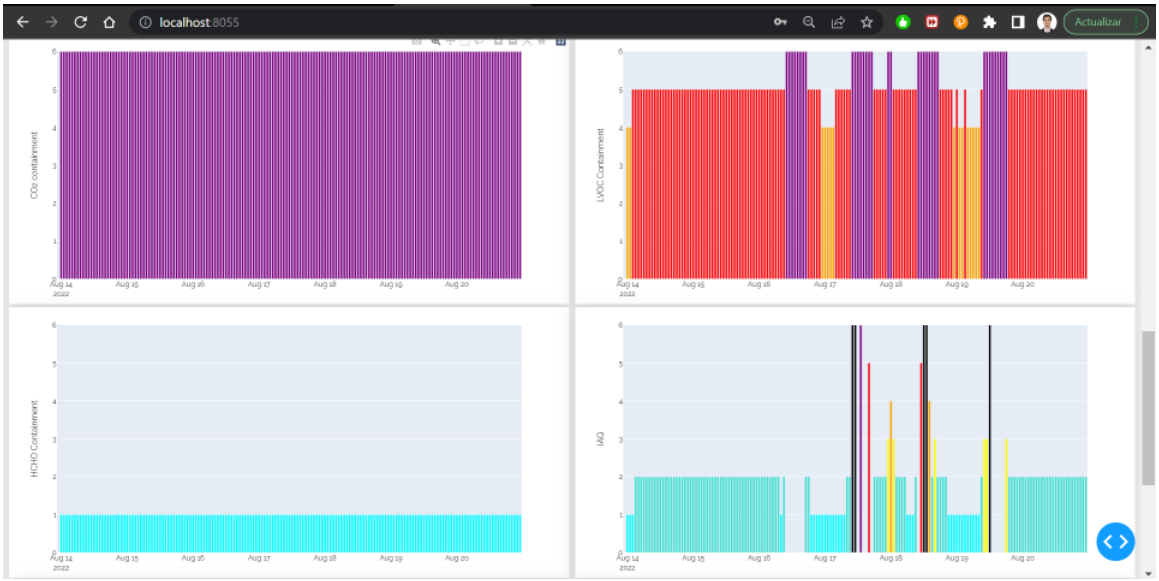


Figure 4.3: Air Quality indicators in the dashboard.

Chapter 5

Building geometry

As the *Ktirio Project* focuses on building's energetic simulations, considering the geometric aspect of the problem can become really usefull.

Previously, the dashboard was capable of grouping sensors by thermal zone or spaces, but it was extremely hard to make use of this without knowing the building's geometry.

Because of this, we decided to add features to the dashboard that will faiclitate the comprehension of the data by visualizing and analysing the building's geometry.

These features all work with IFC files and can be resumed in:

- Creating a 3D interactive and dynamic model of the building, that provides more visual information on the sensors.
- Integrating an IFC analysis tool that links the components and objects of the IFC file.
- Adding solar flux intakes to the building's geometry.

For the dashboard to be secure and avoid conflicts, the uploaded ifc file is never uploaded to the server. When the user uploads the file, the contents are read client-side and stored on the browsers cache.

1 3D View

To be able to show a 3D model of a building using its IFC file, we need to extract each component and its geometry. To do this, we use the python library *ifcopenshell*, which allows us to open ant process .ifc files using python.

Note that the model only uses Plotly for plotting the building. The view can be seen in the figure 5.1.

First, using the *geom* package of the *ifcopenshell* library, we created a function to extract the geometry of an ifc entity. The function returns a dictionnary containing the coordinates of the vertices and indices of the edges and faces of the 3D mesh representing

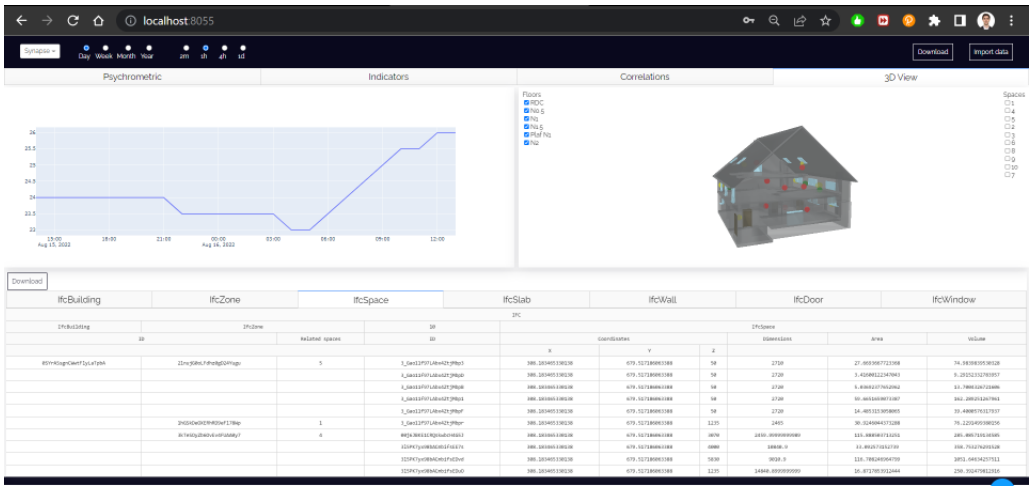


Figure 5.1: 3D view tab of the dashboard.

the entity. Depending on the entity type, the meshes are splitted. This allows to change the color and transparency depending on its material.

Then, we split the building by floors, and get the meshes related to the entities in each floor. This allows the user to select or deselect floors to better visualize certain spaces.

Next, we created a function that highlights thermal spaces, by fetching the IDs of all elements inside each zone.

Finally, the sensors can be visualized in the model by simply fetching their cartesian coordinates.

We also introduced another plot that shows the data of a sensor when selected in the 3D view using the *clickData* property of the Plotly Figure.

2 IFC Analysis Tool

The IFC Analysis Tool, developed by Fabien Allemand, takes an IFC file to diagnose it and visuaize it. The diagnosis verifies that the file correctly describes the geometry of the building. It returns a formatted Microsoft Excel file where each entity and its properties (such as its ID, area and volume) appear.

On the figure 5.2 we can see how this resulting file looks.

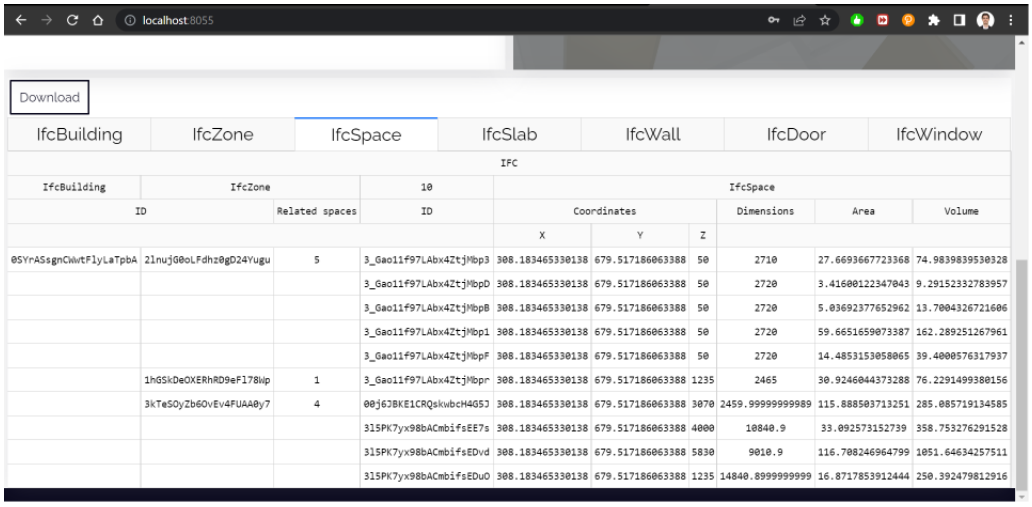
We must note that the provided tool writes the result file onto the server. To avoid conflicting files or populating the server with result files, they are deleted once the Excel file is showed.

An important point is that Plotly is unable to directly display .xlsx files as tables. To solve this, we use Dash tabs. Each sheet of the excel file is read and appended as the tab's content. Also, as some cells of the file are merged (and plotly does not recognize this), they are renamed so that they have the same name. Then, the argument *merge_duplicate_headers* is set to True in the *dash_table*.

	A	B	C	D	E	F	G	H	I	J	K
1	IFC										
2	IfcBuilding	IfcZone		10	IfcSpace						
3	ID	ID	Related spaces	ID	Coordinates			Dimensions	Area	Volume	
4					X	Y	Z				
5	0SYrASsgnCWwtFlyLaTpbA	2lnujG0oLFdhz0gD24Yugu	5	3_Gao11f97LAbx4ZtjMbp3	308.1835	679.5172	50	2710	27.66937	74.98398	
6				3_Gao11f97LAbx4ZtjMbpD	308.1835	679.5172	50	2720	3.416001	9.291523	
7				3_Gao11f97LAbx4ZtjMbpB	308.1835	679.5172	50	2720	5.036924	13.70043	
8				3_Gao11f97LAbx4ZtjMbp1	308.1835	679.5172	50	2720	59.66517	162.2893	
9				3_Gao11f97LAbx4ZtjMbpF	308.1835	679.5172	50	2720	14.48532	39.40006	
10		1hGSkDeOXERhRD9eFl78Wp	1	3_Gao11f97LAbx4ZtjMbpr	308.1835	679.5172	1235	2465	30.9246	76.22915	
11		3kTeSOyZb6OvEv4FUAA0y7	4	00j6JBKE1CRQskwbcH4G5J	308.1835	679.5172	3070	2460	115.8885	285.0857	
12				3l5PK7yx98bACmbifsEE7s	308.1835	679.5172	4000	10840.9	33.09257	358.7533	
13				3l5PK7yx98bACmbifsEDvd	308.1835	679.5172	5830	9010.9	116.7082	1051.646	
14				3l5PK7yx98bACmbifsEDuO	308.1835	679.5172	1235	14840.9	16.87179	250.3925	
15											

Figure 5.2: Excel File resulting from the Diagnosis Tool, for the Synapse building.

We can see how it looks in the figure 5.3 .



IFC

IfcBuilding	IfcZone		10	IfcSpace					
ID		Related spaces	ID	Coordinates			Dimensions	Area	Volume
				X	Y	Z			
0SYrASsgnCWwtFlyLaTpbA	2lnujG0oLFdhz0gD24Yugu	5	3_Gao11f97LAbx4ZtjMbp3	308.183465330138	679.517186063388	50	2710	27.6693667723368	74.9839839530328
			3_Gao11f97LAbx4ZtjMbpD	308.183465330138	679.517186063388	50	2720	3.41600122347043	9.29152332783957
			3_Gao11f97LAbx4ZtjMbpB	308.183465330138	679.517186063388	50	2720	5.03692377652962	13.7004326721606
			3_Gao11f97LAbx4ZtjMbp1	308.183465330138	679.517186063388	50	2720	59.6651659073387	162.289251267961
			3_Gao11f97LAbx4ZtjMbpF	308.183465330138	679.517186063388	50	2720	14.4853153058065	39.4000576317937
	1hGSkDeOXERhRD9eFl78Wp	1	3_Gao11f97LAbx4ZtjMbpr	308.183465330138	679.517186063388	1235	2465	30.9246044373288	76.2291499380156
	3kTeSOyZb6OvEv4FUAA0y7	4	00j6JBKE1CRQskwbcH4G5J	308.183465330138	679.517186063388	3070	2459.99999999989	115.888503713251	285.085719134585
			3l5PK7yx98bACmbifsEE7s	308.183465330138	679.517186063388	4000	10840.9	33.092573152739	358.753276291528
			3l5PK7yx98bACmbifsEDvd	308.183465330138	679.517186063388	5830	9010.9	116.708246964799	1051.64634257511
			3l5PK7yx98bACmbifsEDuO	308.183465330138	679.517186063388	1235	14840.8999999999	16.8717853912444	250.392479812916

Figure 5.3: Diagnoses file shown as a Dash data_table for the Synapse building.

We can see that all the information from the Excel file is in the table on the Dashboard, but unfortunately, colors are lost and some of the cells are still not merged. We hope to fix this issue on the near future.

3 Adding Solar Flux Intakes

By taking weather information and studying the building geometry, the solar radiation on the surfaces can be calculated. The application of these geometric models is worked by Kaisheng Zhang and is planned on being implemented on the dashboard.

This feature is still not implemented at the time of this report due to a lack of time during the internship, but will soon be implemented.

We hope to be able to visualize, once an ifc file and a weather file are provided, the solar flux in each one of the external surfaces of the building. A heatmap will be present

in the 3D view indicating the current radiation value. We also want to be able to plot the historical values of solar radiation when clicking on a surface.

This feature will be extremely useful to link the external weather information to the Psychrometrics of the building. It will help to improve predictions and to better understand some of the models.

Conclusion

In conclusion, we enhanced the dashboard by adding features to make it more optimal, performant, complete, flexible and customizable.

Most of the features exposed in the *objectives* section of this report were added. unfortunately, some of these improvements could not be completed in the course of the internship, but are planned to be worked on in the future.

Concerning the database, the code is currently written to work with csv files as data tables, but can be modified to accept a MySQL schema. The requirements were estimated and the schema was explored.

As for the data imputation, an algorithm that uses a Random Forest regressor to imputate missing information was implemented. Later on, it is expected to explore new algorithms that could lead to better results. Also, we want to be able to predict future data of time periods of multiple lenghts.

We still want to integrate the solar radiations models to the dashboard to facilitate the energetic models comprehension.

Next, as we want the data to be easily understood by the user, we want to improve the visualization part by adding a 3D view of the building in many of the dashboard's tab. By doing this, the user will have a better understanding of the sensor data they are currently viewing, and will provide a tool that is unique to the data provider's dashboards.

Finally, as the dashboard could only be accessed locally at the start of the internship, by starting a Flask server on the terminal and opening in a browser. Because we want to make the platform accessible to multiple companies that could benefit the services provided, it was deployed and available in a web server. The deployment of this tool was managed by Dr. Christophe Prud'homme, who is the head of *CEMOSIS*, using a Docker image that is rebuilt everytime a pull request is merged to the stable branch *main*.

The dashboard can be accessed through http://dashboard.ktirio.fr/

Bibliography

[1] https://www.statistiques.developpement-durable.gouv.fr/bilan-energetique-de-la-france-en-2021-donnees-provisoires-0

[2] https://www.ibm.com/cloud/learn/random-forest

[3] https://dash.plotly.com/pattern-matching-callbacks

[4] https://www.ashrae.org/technical-resources/bookstore/standard-55-thermal-environmental-conditions-for-human-occupancy

[5] https://en.wikipedia.org/wiki/Monte_Carlo_method