**Developers guide to Smart Office Sensor Display (SOSD)**

[Introduction 2](#_Toc490499653)

[Functional Block Diagram 2](#_Toc490499654)

[Database Tables 3](#_Toc490499655)

[Plugins and APIs 5](#_Toc490499656)

[User details & Data download 6](#_Toc490499657)

[User inputs & Chart display 8](#_Toc490499658)

[Data processing 9](#_Toc490499659)

[Google Charts 11](#_Toc490499660)

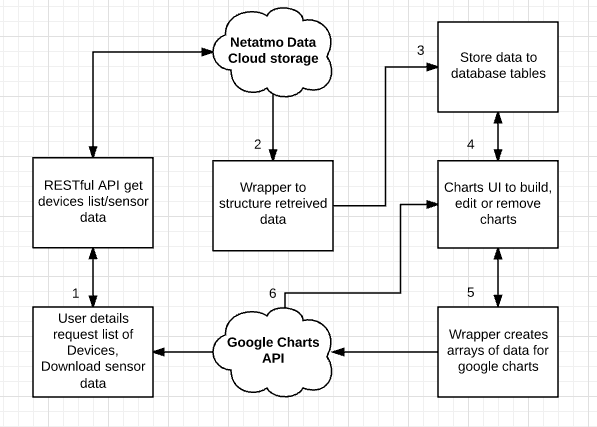
[Default setting for loading three chart types on start-up 14](#_Toc490499661)

[RESTful API example from Netatmo Insight account 15](#_Toc490499662)

# Introduction

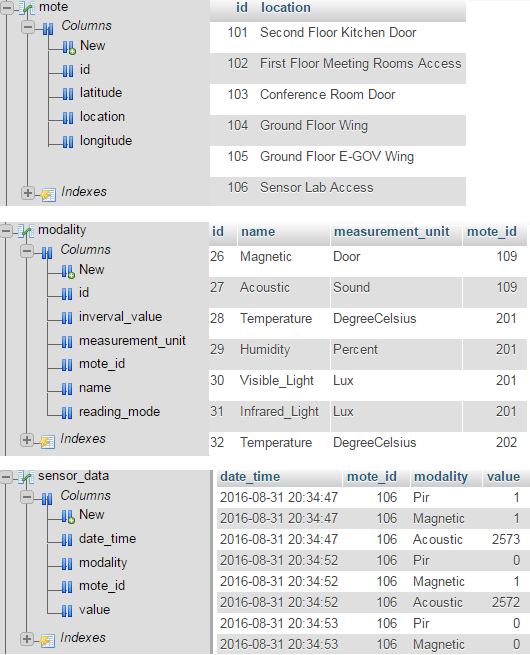
Inside the office building of Insight are some Netatmo sensors spread out across three floors taking different measurements for various environmental readings like Humidity, Temperate, Light, Sound etc. These readings are available on the Netatmo weather app website (https://www.netatmo.com/en-US/product/weather/weatherstation) which Insight is subscribed to. By using an access token this data can also be accessed through a RESTful API returning JSON-LD data. Using this method to store selected historical data to a local mySQL database this project is designed to display that data through a user interface. Different sets of data can be selected with inputs from a user and then displayed in multiple charts for analyses. [Google Charts](https://developers.google.com/chart/) was used to display the data per day as an average or count depending on the sensor. Some functionality was added to allow the user to select a chart type and which sensors to be displayed on a chart. Any number of charts can be added to the page with the option to remove selected charts one by one.

# Functional Block Diagram

****

# Database Tables

Three database tables are used to store the data retrieved in the RESTful API calls to Netatmo. The **mote** and the **modality** tables are accessed when the charts UI page is loaded and the **getMotesLoc.php** script is invoked. This runs two mySQL queries to populate all the motes and there location along with all the different sensors and units to a table in the navigation section to the left. The third table **sensor\_data** is accessed twice when the user hits the ‘Mote Detail & Graph Data’ button. This calls both **get\_detail.php** and **get\_data.php** scripts to retrieve all the data from the selected mote within the selected date range given by the user. The first script creates a table showing a list of sensors and there min/max values for the given dates. The second script pulls all the values logged against each sensor.

****

The following steps show how to create the database and install the tables;

1. Open the Command Prompt on your windows machine. Go to the mysql bin directory. Default is usually “C:\xampp\mysql\bin”
2. Enter the following and hit enter: mysql -h localhost -u root –p

Password is not setup by default. So leave blank or enter your pw if you setup one and hit enter.

1. Once connected run the following commands to set up the database and its tables hitting enter after each one.

1: create database PSOSD;

2: use PSOSD;

The command line should show you are currently in the PSOSD database as follows: “MariaDB [PSOSD]>”

3: create table mote(

id int(11) not null primary key,

location varchar(50)

);

You will see the following notifying you it has run ok “Query OK, 0 rows affected (0.53 sec)”

4: create table modality (

id int(11) not null primary key,

name varchar(50) not null,

measurement\_unit varchar(50),

mote\_id int(11) not null,

constraint fk\_modality\_mode

foreign key(mote\_id) references mote(id)

);

5: create table sensor\_data (

date\_time datetime,

mote\_id int(11),

modality varchar(50),

value double,

constraint fk\_sensor\_mode

foreign key(mote\_id) references mote(id)

);

6: Finally you can exit: exit

# Plug-ins and APIs

**Pace:**

One of the projects specifications was to have some sort of a loading bar to display the progress of a user request. This would include the process of pulling and sorting the data from the local database and also having the data passed into the Google API to display in a chart. The [Pace](http://github.hubspot.com/pace/) library files were downloaded locally for this.

<link href="css/paceBar.css" rel="stylesheet"> <!-- Pace Loading bar -->

<script src="js/pace.js" type="text/javascript"></script> <!-- Pace js library -->

To run the API manually the following code was placed at the start of the main functions **getData()** and **submit()**.

Pace.restart();

**DatePicker:**

To help the user select valid dates for the input fields a [datepicker](https://jqueryui.com/datepicker/) API was used. Once the user clicked inside the date input box a mini calendar would pop up allowing the user to select a date. The format was set to work with the database.

<link rel="stylesheet" href="css/dataPickerAPI-1.12.1.css"> <!-- Date picker -->

<script src="js/dataPickerAPI-1.12.1.js" type="text/javascript"></script> <!-- Date picker -->

// API for the date input fields used for the SQL query, Format set for mySQL.

$.datepicker.setDefaults({

dateFormat: "yy-mm-dd"

});

$(function() {

$("#start\_date").datepicker();

$("#end\_date").datepicker();

});

**Google Charts:**

For displaying the data to the user the Google Charts API was added. The user has to be logged on as the library files cannot be stored locally. The Corecharts package was used for this application to display charts as a bar, line or area chart.

<script type="text/javascript" src="https://www.gstatic.com/charts/loader.js"></script> <!-- Google charts API -->

google.charts.load('current', {'packages':['corechart','controls']});

// Set a callback to run when the Google Visualization API is loaded.

google.charts.setOnLoadCallback(drawChart);

// Callback that creates and populates a data table, instantiates the chart, passes in the data and draws it.

function drawChart() {

......

Bootstrap and jQuery were also added to make a dynamic webpage along with my own JS and CSS files.

<link href="css/bootstrap.css" rel="stylesheet"> <!-- Bootstrap -->

<link href="css/style.css" rel="stylesheet"> <!-- my Styles -->

<script type="text/javascript" src="js/jquery-1.11.3.min.js"></script> <!-- jQuery api -->

<script src="js/bootstrap.js" type="text/javascript"></script> <!-- Bootstrap jQuery -->

<script src="js/myJS.js" type="text/javascript"></script> <!-- My JS file -->

# User details & Data download

The first page on the website requires a user’s access token. This is used in creating a RESTful API that will be sent to Netatmo using the method ‘getstationsdata’ to retrieve all the devices registered for that users token. See the function below ‘**fetchNetatmoDevices**’.

//Get device data from netatmo via users access token\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

function fetchNetatmoDevices() {

var getstationsdata = "https://api.netatmo.com/api/getstationsdata?get\_favorites=true&access\_token=";

token = $('#token').val();

getstationsdata += token;

var request = new XMLHttpRequest();

request.onreadystatechange = function() {

//4:request finished and response is ready / 200: "OK"

if (request.readyState == 4 && request.status == 200) {

callback(request.responseText);

} else if(request.readyState == 4 && request.status == 403) {

alert(request.status+request.responseText);//alerts if access token failed

}

};

request.open('GET', getstationsdata, true);

request.send();

};

On successful retrieval of the data, it is then structured in the **callback** function. Here a list of devices and sensors along with their setup and last data record dates are stored in a variable called ‘**floors’**. This variable is used to populate a table to the webpage for the user to see. It is also used to pass the data in JSON format to a php script where is it stored to the local database via ‘**insertDevices.php**’. Here the table’s **mote** and **modality** are populated.

The next step is for the user to select valid start and end dates to download the sensor data sets. For this the ‘getmeasure**’** method is used in a RESTful request to Netatmo. As this request needs to be sent for each device an object array of AJAX requests are created as **xhr[x]** in the function ‘**fetchNetatmoValues’**. See the code snippet below for reference. The object **obj** holds the device ids as a property and the URL as the value for each device. This is used to create the xhr Ajax requests.

for( var i=0;i<floors.length;i++) {

deviceURL = "&device\_id="+floors[i].Device\_id+"&type="+floors[i].Sensors;

moteId = floors[i].DevicePK;

urlObj[moteId] = (getmeasure+deviceURL);

moduleURL = "&device\_id="+floors[i].Device\_id+"&module\_id="+floors[i].Module\_id0+"&type="+floors[i].Module\_Sensors0;

moteId = floors[i].Module\_PK;

urlObj[moteId] = (getmeasure+moduleURL);

}

urlArray.push(urlObj);

xhr =[];

var obj = urlArray[0];

urlKeys = Object.keys(obj);

for(var i=0;i<urlKeys.length;i++) {

var url = urlArray[0][urlKeys[i]];

xhr[i] = new XMLHttpRequest();

xhr[i].open('GET', url, true);

}

The first RESTFul API is sent and once ready calls a function ‘**sensorDataWrapper**’ to structure the returned data set and store it in the **sensor\_data** table via the php script ‘**insertSensorData.php’**.

//run the first xhr request.

xhr[0].onreadystatechange = function() {

//4:request finished and response is ready / 200: "OK"

if (xhr[0].readyState == 4 && xhr[0].status == 200) {

sensorDataWrapper(xhr[0].responseText, urlKeys);

} else if(xhr[0].readyState == 4 && xhr[0].status == 403) {

alert(xhr[0].status+xhr[0].responseText);//alerts if access token failed

}

};

In order to loop through each Ajax request a counter is incremented, starting from zero, once a devices data set is processed and stored. This counter is then used to call the next Ajax request in the **xhr[x]** object array. As each one is populated to the table a message is sent to the webpage to inform the user. The function **ajaxLoop** is called after each Ajax request is completed to trigger the next one.

count++

if(count<xhr.length) {

xhr[count].send();

ajaxLoop(count);

} else { ......

// loop each xhr request

function ajaxLoop(x) {

Pace.restart();

xhr[x].onreadystatechange = function() {

try {

//4:request finished and response is ready / 200: "OK"

if (xhr[x].readyState == 4 && xhr[x].status == 200) {

sensorDataWrapper(xhr[x].responseText, urlKeys);

} else if(xhr[x].readyState == 4 && xhr[x].status == 403) {

alert(xhr[x].status+xhr[x].responseText);//alerts if access token failed

}

}

catch(e) {

alert("Error! "+xhr[x].status+": "+xhr[x].responseText+"\nPlease try again")

}

};

};

Once all the data sets are downloaded and the user is happy to proceed, a button is made available for the user to navigate to the charts.html page where he/she can view, edit, create and remove charts.

# User inputs & Chart display

The main page is made up of two sections, the side menu on the left and the charts display section. The side menu can be toggled from view using the two functions **openNav()** and **closeNav()** which are invoked when the user hits the **open** or **close** icons. At the bottom of the side menu the user can view two tables. The first table is a reference of all the sensors and there measurement units. The second table contains a list of the available motes and there locations. This is populated by the script, **getMotesLoc.php**, which is run when the page is loaded. It pulls the data located in the local database table’s **modality** and **mote** which was populated when the RESTful API retrieved the raw data from Netatmo.

There are some inputs for the user to select what mote and chart they would like to display. These values are stored in JavaScript as global variables to be referenced in different functions. Also the mote id, start and ends dates are stored as cookies to be used in the various php scripts.

There are three buttons for the user to select. At the start only the ‘Mote Detail & Graph Data’ button is displayed. Once all the required fields are populated this button will run two scripts, **get\_detail.php** and **get\_data.php**, to gather and sort the selected data. Once the data is retrieved successfully two buttons will be displayed for the user to select, ‘Draw Graph’ or ‘Reset’. If the user chooses the ‘Reset’ button then he/she will start the selection process again. If the user hits the ‘Draw Graph’ button then the data will be sent to the Google API through the **submit()** function. Here the data is displayed on the page as a chart along with a details table for the given set of data showing the min/max values and sensors linked to the mote id.

Each chart when created is assigned a unique variable id ‘graphCount’ to its HTML Div container. This variable id is incremented after each chart is populated to insure uniqueness. It is also used as a parameter for the function **removeDiv()** which is linked to each charts Div when created. This allows a user to remove a chart of their choosing. As each chart is contained in the Div class ‘row’, any charts below the deleted chart will be pushed up in its place.

# Data processing

In order for the Google charts API to work the data had to be wrapped into arrays. One array is created for the dates array and one for each sensor associated with the mote id. In order to represent the data in a meaningful way an average was collected for all the sensor types, except the event/counter sensors, for each day and then this average was assigned to its sensor array. Therefore each element in the arrays represented an average for the given sensor for a given day. The array lengths were determined by the amount of days in the user selected date range. This process is done in the http request **dataReq.php**. The object ‘graphArrays’ was used to store all the sensor arrays. If a sensors max value was greater than one then an average was needed. Three arrays were created using the sensor property from the motedetail array of objects. If the max value was not greater then one then this was an event counter and so just a sensorCount array was created.

// Create object with properties named after each sensor with a value of an array of length equal to the num of days in the user date range.

graphArrays = {}; // reset needed for each new graph.

for(var i=0;i<motedetail.length;i++) {

if(motedetail[i].MaxVal>1) { // If value is greater than 1 then we work out an average by having sum and avg array also

graphArrays[motedetail[i].sensor] = new Array(diffDays).fill(0);

graphArrays[motedetail[i].sensor+'Sum'] = new Array(diffDays).fill(0);

graphArrays[motedetail[i].sensor+'Avg'] = new Array(diffDays).fill(0);

}// if value is not greater than one we use this sensor as a counter only (An event reading)

else { graphArrays[motedetail[i].sensor+'Count'] = new Array(diffDays).fill(0); }

}

Motedetail is an array of objects containing the details for each sensor in the mote. This allows the method to be applied to any system with the same database setup. An example of the details captured for mote 101 is shown below;

motedetail = (3) [Object, Object, Object]

1. Object
   1. MaxVal:"1"
   2. MinVal:"0"
   3. sensor:"Pir"
2. 1:Object
   1. MaxVal:"1"
   2. MinVal:"0"
   3. sensor:"Magnetic"
3. 2:Object
   1. MaxVal:"2512"
   2. MinVal:"2118"
   3. sensor:"Acoustic"

The next process is the most intensive as each entry in the database is stored as an object to the variable ‘moteArray’. This array of objects can contain thousands of entries depending on the date range selected by the user. Each object contains three properties as shown in example below;

1. Object
   1. date\_time:"2016-09-01 09:56:43"
   2. observation:"1"
   3. sensor\_type:"Pir"

The following process will loop through each object by date, sensor name and value. Depending on these values the ‘graphArrays’ objects will be updated for each sensor where the element position matches the current day in the loop.

//Loop through every object in the array to get a count per sensor per day

moteArray.forEach(function(obj) {

var d=new Date(startDate); // set startDates date obj as d for loop below

for(var j=0; j<diffDays; j++) { // \*\*\*\*\*\*\*\*\*\*\*\*\*\* loop every day \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for(var x=0;x<motedetail.length;x++) { //loop each sensor name from the motedetail object

if(new Date(obj.date\_time).getDate() ===d.getDate() && obj.sensor\_type===motedetail[x].sensor && obj.observation==="1") {

graphArrays[obj.sensor\_type+'Count'][j]++; //increment the day count array for the current sensor property. ie. Pir on 2nd day => obj {Pir: [0,1,0,0], Mag...:[0,0,0,0], ....}

} else if(new Date(obj.date\_time).getDate() ===d.getDate() && obj.sensor\_type===motedetail[x].sensor && obj.observation!=="0") {

graphArrays[obj.sensor\_type][j]++;

graphArrays[obj.sensor\_type+'Sum'][j] += parseInt(obj.observation);

parseFloat(graphArrays[obj.sensor\_type+'Sum'][j].toFixed(2)); //Convert Sum arrays to float(2)

}

}

d.setTime(d.getTime()+oneDay); // increment the date by one day

};

});

Finally all that is needed is to calculate the average and update the array for each sensor. This is calculated by dividing each sum arrays element by its corresponding arrays counter element.

//\*\*\*\*\*\*\*\*\*\*\* Calc the avg per day

for(var i=0;i<diffDays;i++) { //diffDays is equal to the length of graphArrays

for(var x=0;x<motedetail.length;x++) {//loop through each sensor

if(graphArrays[motedetail[x].sensor+'Sum']) { //Check if this sensor has a sum array

avg = parseInt(graphArrays[motedetail[x].sensor+'Sum'][i]/graphArrays[motedetail[x].sensor][i]);

if(!isNaN(avg)) {

graphArrays[motedetail[x].sensor+'Avg'][i] = avg;

} else { graphArrays[motedetail[x].sensor+'Avg'][i] = 0 }

}

}

};

# Google Charts

As mentioned earlier Google charts was used to display the sensor data to the user. Here is a step through guide to explain the process.

First the corecharts package is loaded. This contains the library files for the charts we want to use. Once this has loaded a callback function needs to be declared. For this project the callback function is called **drawChart()**.

google.charts.load('current', {'packages':['corechart','controls']});

// Set a callback to run when the Google Visualization API is loaded.

google.charts.setOnLoadCallback(drawChart);

The DataTable object is used to hold the data passed into a visualization. A DataTable is a basic two-dimensional table. All data in each column must have the same data type. Below we set this object to a variable **data**. Also the date column and the sensor columns are added for average and counter sensors by using the **addColumn()** method. Another variable **row** is assigned to hold the array values for each column. Each row variable is then added to the data object using the **addRow()** method.

function drawChart() {

var data = new google.visualization.DataTable();

data.addColumn('date', 'Date');

for(var sensors in graphArrays) {

if(sensors.includes('Avg') | sensors.includes('Count')) {

data.addColumn('number', sensors);

}

};

//reset startDate object from input

startDate = new Date($('#start\_date').val());

for(i=0;i<diffDays;i++) {//loop each sensor array item

var row = [new Date(startDate)]; // First col been the date

for(sensor in graphArrays) { // Loop each sensor value to add to chart col

if(sensor.includes('Avg') | sensor.includes('Count')) { //Ignore sum and counter for avg

row.push(graphArrays[sensor][i]);

}

}

data.addRow(row);

startDate.setTime(startDate.getTime()+oneDay); // increment the startDate object by mil sec so month change is made

};

To add some functionality to the user a dropdown list was created with each chart listing all the columns bar the date column i.e. the sensors. This was done by creating a new ‘visualization.DataTable()’ called **columnsTable**.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* column selector \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

var columnsTable = new google.visualization.DataTable();

columnsTable.addColumn('number', 'colIndex');

columnsTable.addColumn('string', 'colLabel');

var initState= {selectedValues: []};

// put the columns into this data table (skip column 0 - Date)

for (var i = 1; i < data.getNumberOfColumns(); i++) {

columnsTable.addRow([i, data.getColumnLabel(i)]);

initState.selectedValues.push(data.getColumnLabel(i));

}

Now that the charts **data** object and the column filter **columnsTable** object are created we need to create a container or a wrapper to hold the data and set its location in HTML with some visual options.

var columnFilter = new google.visualization.ControlWrapper({

controlType: 'CategoryFilter',

containerId: 'colFilter\_div'+graphCount, // sets new div for each graph

dataTable: columnsTable, // DataTable() object

options: {

filterColumnLabel: 'colLabel',

ui: {

label: '',

caption: "Select Sensors", //Button Display

allowTyping: false,

allowMultiple: true,

allowNone: false,

selectedValuesLayout: 'aside'

}

},

state: initState

});

var chart = new google.visualization.ChartWrapper({

chartType: chartType, // User selected radio button

containerId: 'chart\_div'+graphCount,

dataTable: data, // DataTable() object

options: {

title: title, // Title variable set at start of submit func

width: '100%',

height: 200,

pointSize: 3, // For chart display visual

hAxis: {

title: 'Date(Month/date/year)',

format: 'MMM/d/yy EEE' //show date format as ex Sep/4/16 Sun

},

vAxes: {

0: {title:'Sensor counter & Avg'}

},

explorer: {

axis: 'horizontal'

}

}

});

Before the above DataTable() views are created we need to set the new HTML to hold the data. The **removeDiv()** function is assigned to the unique Div container along with the column filter and chart. The details table mentioned earlier is added beside the chart and the ‘graphCount’ incremented for the next chart.

//create new Divs to hold the graph filter and chart

var newDom = "<div id='"+graphCount+"' class='row chart'><div id='contentDetail"+graphCount+"' class='col-md-3'></div>";

newDom += "<div class='col-sm-7 col-md-8'><span class='glyphicon glyphicon-remove' style='float: right;' onclick='removeDiv("+graphCount+")'></span>";

newDom += "<div id='colFilter\_div"+graphCount+"'></div><div id='chart\_div"+graphCount+"'></div></div></div>";

//add the Dom to the index page by added it to the graph container div.

$('.graphContainer:last').append(newDom);

$('#contentDetail'+graphCount).html(detailContent);

graphCount++; // increment the counter for the next graph

An event listener is added to the columnFilter so when the user changes the selection of sensors to be displayed the chart is redrawn to the page by invoking the function **setChartView()**. The columnFilter is then redrawn again to show only the sensors that are not selected.

google.visualization.events.addListener(columnFilter, 'statechange', setChartView);

setChartView(); //draw chart

columnFilter.draw(); // draw the column filter

} // end of DrawChart func

This function loops through the selected values in the columnFilter object and adds only those columns to a **view** variable which is then used to draw the chart.

function setChartView () {

var state = columnFilter.getState();

var row;

var view = {

columns: [0]

};

for (var i = 0; i < state.selectedValues.length; i++) {

row = columnsTable.getFilteredRows([{column: 1, value: state.selectedValues[i]}])[0];

view.columns.push(columnsTable.getValue(row, 0));

}

// sort the indices into their original order

view.columns.sort(function (a, b) {

return (a - b);

});

chart.setView(view);

chart.draw();

}

# Default setting for loading three chart types on start-up

When the user enters the user interface webpage the application is setup so that three default graphs are drawn up showing the user a bar chart, Area chart and a Line chart over five days. From the code below you can see the defaultStartDate variable is first checked. This is set when new data is retrieved from Netatmo. If this is the first time the application is been run then the defaultStartDate will not be set and the test data will be loaded still in the local database. The defaultStartDate and DefaultEndDate are then set to the test data entry 2016-09-01 for five days. The cookies are also set for the php scripts to fetch the data from the local database.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Setup default display of three graph types for 5 days on the first mote

//First set the dates and the mote as the first from the list.

if(defaultStartDate) { //If start Date set for restful request set default endDate to + five days.

defaultEndDate = new Date(defaultStartDate);

defaultEndDate.setDate(defaultEndDate.getDate()+5);

document.cookie = "startDate="+defaultStartDate.getFullYear()+"-"+(defaultStartDate.getMonth()+1)+"-"+defaultStartDate.getDate()+";";

document.cookie = "endDate="+defaultEndDate.getFullYear()+"-"+(defaultEndDate.getMonth()+1)+"-"+defaultEndDate.getDate()+";";

document.cookie = "mote="+firstMote+";";

diffDays = Math.round(Math.abs((defaultStartDate.getTime() - defaultEndDate.getTime())/(oneDay))); //Get the # of days from start and end dates

} else { // Normally a error will be posted to the user and no default data shown. Still in development mode as no data from RESTful API as yet, instead from DB So manually entering following dates

defaultStartDate = new Date('2016-09-01');

defaultEndDate = new Date(defaultStartDate);

defaultEndDate.setDate(defaultEndDate.getDate()+5);

document.cookie = "startDate=2016-09-01;";

document.cookie = "endDate=2016-09-06;";

document.cookie = "mote="+firstMote+";";

diffDays = Math.round(Math.abs((defaultStartDate.getTime() - defaultEndDate.getTime())/(oneDay))); //Get the # of days from start and end dates

}

Once the php scripts have retrieved the data the Google library is loaded and three instances are created for three callback functions. Each set to draw one of three charts.

google.charts.load('current', {'packages':['corechart','controls']});

// Loop three times to draw each chart type.

// Set a callback to run when the Google Visualization API is loaded.

google.charts.setOnLoadCallback(drawDefaultChart1); // Bar chart

google.charts.setOnLoadCallback(drawDefaultChart2); // Area chart

google.charts.setOnLoadCallback(drawDefaultChart3); // Line chart

# RESTful API example from Netatmo Insight account

Insight-OpenIoT Ground Floor

Insight Welcome Hall & Internet of Things Sensors Lab

70:ee:50:12:9a:5c

Insight-OpenIoT 1st Floor

OutDoor Museum & Museum Hall

70:ee:50:03:96:e0

Insight-OpenIoT 2nd Floor

OutDoor Reception and Kitchen Room

70:ee:50:03:53:1a

For device id 70:ee:50:03:96:e0

https://dev.netatmo.com/resources/technical/reference/weatherstation/getstationsdata

* body:
  + [▼](https://dev.netatmo.com/resources/)devices:
    - [▼](https://dev.netatmo.com/resources/)
      * \_id: "70:ee:50:03:96:e0"
      * cipher\_id: "enc:16:cqEytYU1JOkEVIeJukhynC4Z0hJaY8/GU661P39Y9q35Zp0OQdvR9iAvf8YTw1bB"
      * last\_status\_store: 1497442706
      * [▼](https://dev.netatmo.com/resources/)modules:
        + [▼](https://dev.netatmo.com/resources/)

\_id: "02:00:00:03:94:b4"

type: "NAModule1"

last\_message: 1480305313

last\_seen: 1480305268

[▼](https://dev.netatmo.com/resources/)dashboard\_data:

time\_utc: 1480300602

Temperature: 9

Humidity: 80

date\_max\_temp: 1480291478

date\_min\_temp: 1480296297

min\_temp: 9

max\_temp: 9.6

[▼](https://dev.netatmo.com/resources/)data\_type:

"Temperature"

"Humidity"

module\_name: "Outdoor Museum"

last\_setup: 1440767716

battery\_vp: 3420

battery\_percent: 0

rf\_status: 43

firmware: 43

* + - * [▼](https://dev.netatmo.com/resources/)place:
        + altitude: 17
        + city: "Galway"
        + country: "IE"
        + timezone: "Europe/Dublin"
        + [▼](https://dev.netatmo.com/resources/)location:

-9.0742141297608

53.289732768747

* + - * station\_name: "Insight-OpenIoT 1st Floor"
      * type: "NAMain"
      * read\_only: true
      * [▼](https://dev.netatmo.com/resources/)dashboard\_data:
        + AbsolutePressure: 1008.2
        + time\_utc: 1497442687
        + Noise: 43
        + Temperature: 20.9
        + temp\_trend: "stable"
        + Humidity: 58
        + Pressure: 1010.2
        + pressure\_trend: "down"
        + CO2: 567
        + date\_max\_temp: 1497439960
        + date\_min\_temp: 1497409355
        + min\_temp: 20.3
        + max\_temp: 21
      * [▼](https://dev.netatmo.com/resources/)data\_type:
        + "Temperature"
        + "CO2"
        + "Humidity"
        + "Noise"
        + "Pressure"
      * co2\_calibrating: false
      * date\_setup: 1440768136
      * last\_setup: 1440768136
      * module\_name: "Museum Hall"
      * firmware: 102
      * last\_upgrade: 1440354875
      * wifi\_status: 62
  + [▼](https://dev.netatmo.com/resources/)user:
    - mail: "huy.levan@insight-centre.org"
    - [▼](https://dev.netatmo.com/resources/)administrative:
      * country: "IE"
      * reg\_locale: "en-US"
      * lang: "en-US"
      * unit: 0
      * windunit: 0
      * pressureunit: 0
      * feel\_like\_algo: 0
* status: "ok"
* time\_exec: 0.048340082168579
* time\_server: 1497442817