Load monitoring and control by using Arduino, Raspberry Pi and IoT

Analysis of the different available technologies and examples to help students to develop the Arduino Project.

SESSION 2: WEB CLIENT. WORKING WITH REE REST API.

Guidelines for data acquisition from external sources:

BICING and REE

Content

Introduction

The main goal to achieve at the end of the session is to learn how to consult data from the REE (Red Eléctrica de España) API to obtain the hourly energy prices in the day-ahead market (DAM).

To achieve this objective, the practice is divided into two different parts. First, instead of working directly with the REE API, the *Bicing* API will be presented and used, from the public bike service from Barcelona City Hall. This API is less complex than the REE API and it will be useful as an example before dealing with the second part.

Once the student has worked with and manipulated the python script of the *Bicing* API, the manual will develop the basics points to operate with the REE API.

Bicing API

The city of Barcelona has a public service called *Bicing* where the user can take a bike from a street station for transportation. At the website of http://opendata-ajuntament.barcelona.cat/en/ you can find the documentation of the City Hall basic services. To develop the API session, we will use the next link:

http://opendata-ajuntament.barcelona.cat/data/en/dataset/bicing

In this link, the data related to the Bicing stations of the city of Barcelona is shown. To access this information, the user can work with the JSON file (JavaScript Object Notation) using the following link:

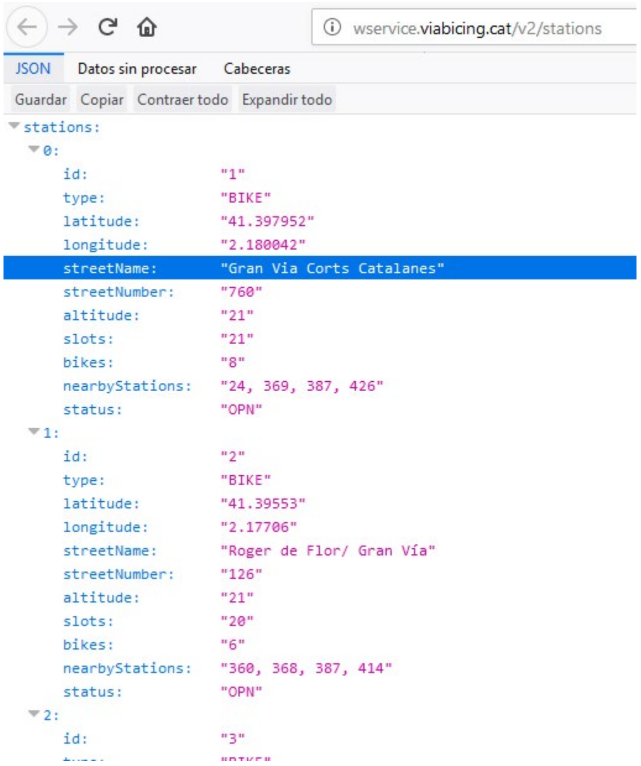
http://wservice.viabicing.cat/v2/stations

The first step is to access this link by means of a browser to visualize the information in a JSON format as it is shown in the screenshot below (Figure 1).

JSON format

Once the JSON file is uploaded in the browser, the information can be consulted in a friendly format. It is important to remark how the information is classified in a JSON file. In the picture of the Bicing JSON, the information classified as the element "stations" can be observed. This class contains the data of all the Barcelona Bicing stations. Inside the "stations" element, there are sub-elements as "id," "streetName," "streetNumber," "bikes," etc.

For example, the first station in the JSON file is located at Gran Via de les Corts Catalanes Avenue, number 760. It has 8 bikes available at the moment of the consultation.



```
stations:
  0:
    id:               "1"
    type:             "BIKE"
    latitude:         "41.397952"
    longitude:        "2.180042"
    streetName:       "Gran Via Corts Catalanes"
    streetNumber:     "760"
    altitude:         "21"
    slots:            "21"
    bikes:            "8"
    nearbyStations:   "24, 369, 387, 426"
    status:           "OPN"
  1:
    id:               "2"
    type:             "BIKE"
    latitude:         "41.39553"
    longitude:        "2.17706"
    streetName:       "Roger de Flor/ Gran Vía"
    streetNumber:     "126"
    altitude:         "21"
    slots:            "20"
    bikes:            "6"
    nearbyStations:   "360, 368, 387, 414"
    status:           "OPN"
  2:
    id:               "3"
```

Figure 1: Obtained JSON file from BICING API

It is important to remark that by means of a single consultation, it is possible to obtain all the elements data in a single JSON file. Furthermore, with this file already loaded, it is possible to consult all the information without having to do new petitions.

Request library

In the previous section, we have visualized the JSON file using the web browser. However, for the development of your project, the information has to be obtained and treated using a script executed in your computer or Raspberry Pi.

An easy way to do this is by using Python language. Python is a powerful programming language with a great variety of libraries and code repository that makes easier the programming task. To obtain the API data information, a REST petition to the API located at the computer server should be done, and one option is to use the Request library.

Requests is a simple and well document HTTP library for Python. The link to the library documentation is the following one:

http://docs.python-requests.org/en/master/api/

All the Requests' functionalities can be accessed by 7 methods. They all return an instance of the Response object. These 7 methods are the following ones:

- `requests.request`(*method*, *url*, ***kwargs*)
  - Constructs and sends a `Request`.
- `requests.head`(*URL*, ***kwargs*)
  - Sends a HEAD request.
- `requests.get`(*URL*, *params=None*, ***kwargs)*
  - Sends a GET request.
- `requests.post`(*URL*, *data=None*, *JSON=None*, ***kwargs*)
  - Sends a POST request.
- `requests.put`(*URL*, *data=None*, ***kwargs*)
  - Sends a PUT request.
- `requests.patch`(*URL*, *data=None*, ***kwargs*)
  - Sends a PATCH request.
- `requests.delete`(*URL*, ***kwargs*)
  - Sends a DELETE request.

As it can be seen, the *request.request()* method is the most generic one, and it can generate a request of any type. The following methods are specific for the specific type of REST requests introduced in the preliminary documentation of this session.

The most common and most used method in REST architecture is GET. For this reason, the present document will develop in more detail only the *request.get()* method.

GET method

The head of the Request.get() method is the next one:

`requests.get`(*url*, *params=None*, ***kwargs*)

where the parameters are:

Parameters:	• **URL** – URL for the new Request object. • **params** – (optional) Dictionary, list of tuples or bytes to send in the body of the Request. • ****kwargs** – Optional arguments that request takes.
Returns:	Response object
Return type:	requests.Response

The most important parameter is the *URL* variable. The variable *params* will be optional in some APIs. This method will return the user a Response object with all the information stored inside.

Response object

If the previous GET method petition is correct, the server will answer to this petition. All the received information is stored in the Response object, which contains a server's response to an HTTP request.

The Response object has a lot of variables and methods. The ones that will be used during these laboratory sessions are:

- `response.status_code`
  - Integer Code of responded HTTP Status, e.g. 404 (not found), 400 (bad request), or 200 (ok).
  - For further details: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes
- `response.url`
  - Final URL location of Response.
- `response.content`
  - Content of the response, in bytes
- `response.json()`
  - Returns the JSON-encoded content of a response, if any

Please bear in mind that, at the programming design level, the correct way to work is with DTO, Data Objects, such as the Response object. This has the main objective for avoiding the saturation of the network with multiple petitions. The idea is to use the highest bandwidth communication to obtain all the possible information from a single petition, instead of doing multiple small petitions to access to small pieces of information. Furthermore, some APIs can ban our access to them such as the Instagram API. This API limits the number of petitions to 5000 with a single user ID in a time period to prevent an excess of petitions.

The computer resources allow us to obtain all the information with a single petition and store the received data in the local memory. Once the information is stored, it will be possible to search and process the desired information.

Bicing API code example

Once introduced the Response library, the student can enter the Learnify platform to download the code example of the Bicing API. The python script is named *requestUsage.py*

The code is written following the style guide that you all have on Learnify. The code is subdivided into different methods and subfunctions. Each subfunction develops a small task. By doing this, reading and understanding the code becomes easier.

The first step is to import all the needed Python libraries. For this example, the only needed library is *requests*.

```python
import requests
```

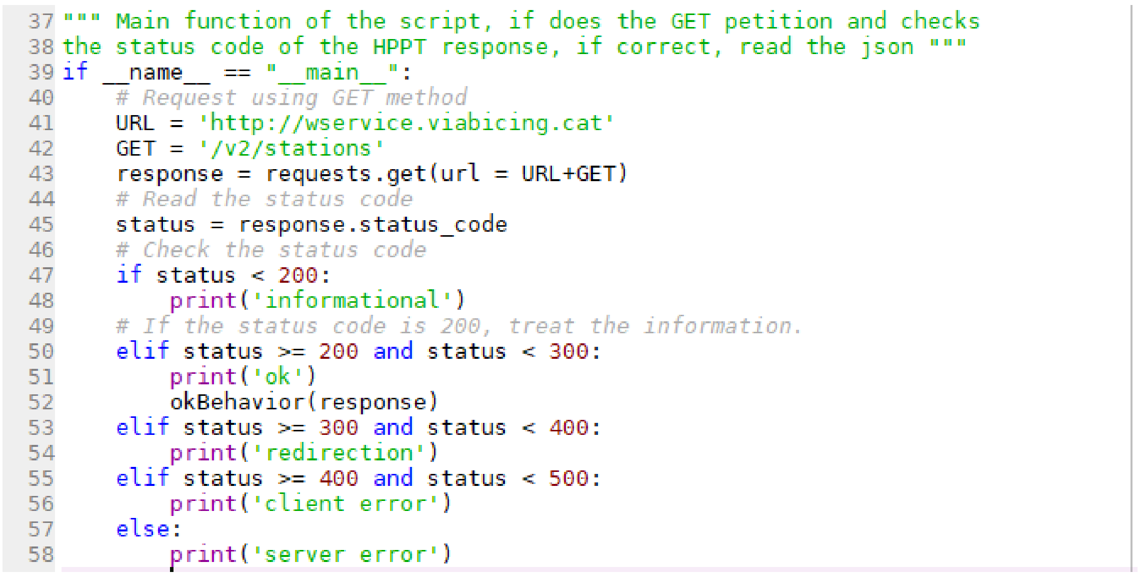
After importing the libraries, the usual is to implement all the subfunctions. At the end of the file, write the main function. For understanding purposes, the manual explains first the main function because it is the first one to be executed by the computer.

The main function is divided into 2 parts. The first one sends the request to the Bicing API using a GET method. The second part reads the answer of the server and acts depending on the answer.

In the first section, we explained how to access the JSON information. The website used for this purpose is the following one:

http://wservice.viabicing.cat/v2/stations

This URL can be divided into two different parts, being the first one the main domain (http://wservice.viabicing.cat) and the second one the resource (/v2/stations).



```python
37 """ Main function of the script, if does the GET petition and checks
38 the status code of the HPPT response, if correct, read the json """
39 if __name__ == "__main__":
40     # Request using GET method
41     URL = 'http://wservice.viabicing.cat'
42     GET = '/v2/stations'
43     response = requests.get(url = URL+GET)
44     # Read the status code
45     status = response.status_code
46     # Check the status code
47     if status < 200:
48         print('informational')
49     # If the status code is 200, treat the information.
50     elif status >= 200 and status < 300:
51         print('ok')
52         okBehavior(response)
53     elif status >= 300 and status < 400:
54         print('redirection')
55     elif status >= 400 and status < 500:
56         print('client error')
57     else:
58         print('server error')
```

In line 41 and 42 we store the *url* divided into two variables as strings. The variable URL and the variable GET.

The Bicing API doesn't need any additional header or parameter, so at line 43 the GET method is used only to introduce the *url* parameter. The *url* is the recomposed (URL + GET) address explained before. The response object returned by the GET method is stored at the *response* variable.

The second part of this main function is to check the answer of the server. We extract the status code returned by the server and store the value into the variable *status* using *response.status_code.*

Usually, if the answer is correct, the status code is 200. Otherwise, the usual answer is a 400. In the first case, the *okBehavior()* method will be called. This method is programmed by us and will treat the information of the *response* object introduced as a parameter.

```
 4 """ If the GET petition is answered corretly, treat the received information """
 5 def okBehavior(response):
 6     data = response.json()
 7     # Use a method to extract from the json the data of the stations
 8     listOfStations = getArrayOfStations(data)
 9     # For each station of the list of all the stations, read the Stret name,
10     # number, state and number of bikes
11     for station in listOfStations:
12         print("station of "+station['streetName']+" "+station['streetNumber']
13         + " is "+station['status']+" and has "+station['bikes']+" bikes")
14     # Use a method to sum all the total available bikes in all the stations
15     print ("the total of bikes is "+totalOfBikes(listOfStations))
```

To define a subfunction, we use the command *def*. This subfunction has as an input parameter the object *response*. The first step done in line 6 is to extract the JSON stored in *response* and store it in the *data* variable.

To treat the desired information obtained from the JSON file, the first step will be to look for the desired elements of the JSON, using the subfunction *getArrayOfStations().*

```
17 """ Method to check if the json contains 'stations' information and return
18 a array with all the data """
19 def getArrayOfStations(jsonData):
20     # Check the elements in the json
21     for element in jsonData:
22         # If there is an element called 'stations', return the element.
23         # Otherwise, return amb empty array
24         if element == 'stations':
25             return jsonData[element]
26     return []
```

This *getArrayOfStations()* subfunction does a search in the JSON variable (jsonData) looking for all the elements. If there is an element called 'stations', the function will return the data of this element as an array.

Returning to the *okBehavior()* subfunction, all the data of the stations are stored in the *listOfStations* array. Using a *for* in line 11, the script prints the following data of the stations:

- Street Name.
- Street Number.
- Status of the station.
- Number of bikes available at the station.

To finish the script, we call a subfunction named *totalOfBikes()* that searches and sums all the available bikes in all the stations.

```python
""" Method to calculate the total number of available bikes """
def totalOfBikes(listOfStations):
    sum = 0
    # For each station, read the element 'bikes' and add it to the total
    for station in listOfStations:
        sum += int(station['bikes'])
    # Return as string the 'sum' variable
    return str(sum)
```

The main objective of this script is to illustrate how to work with a JSON file. This kind of files are too large, so the correct way is to parse the string information into a more manageable array using the *response.json()* method. Once the JSON is loaded in an array and we know the JSON structure, it is possible to look for the desired information using *for* commands.

REE API

Spanish market tariffs

Before entering the API documentation, it is important to understand what the desired information we want to consult is. In the preliminary documentation, the day-ahead electricity auctions mechanism of the Spanish electricity market have been explained.

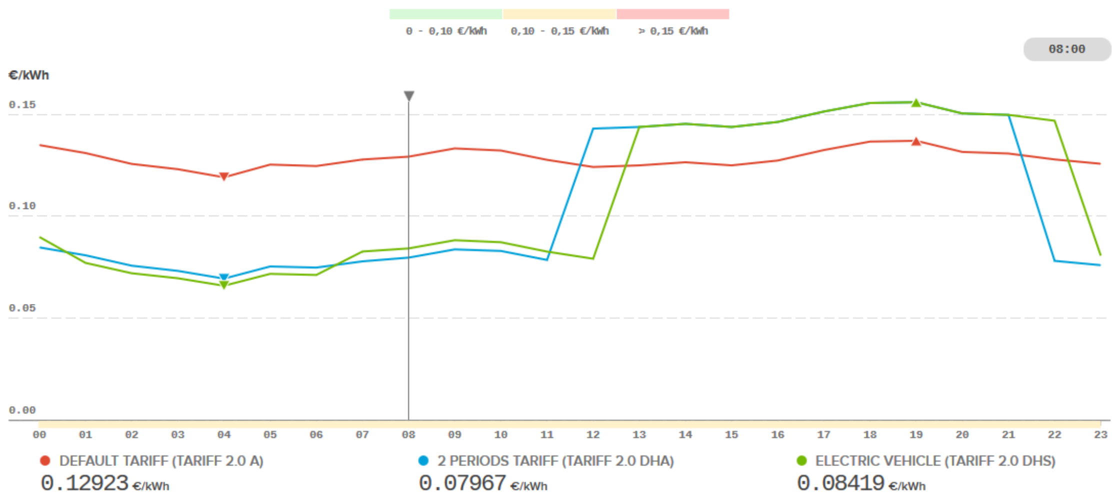
Auctions determine the prices of the energy that will be paid by our electricity retailer. However, the price paid by the user will depend on the contracted tariff. In Spain, for standard end-users three different tariffs are available:

- Default tariff (2.0 A)
- 2 periods tariff (2.0 DHA)
- Electric vehicle tariff (2.0 DHS)

To consult the daily price of these tariffs, we can access the REE webpage. To consult the price of November, 17th 2018 for example, we can access this link:

https://www.esios.ree.es/en/pvpc?date=17-11-2018

There, the user can check the price of the 3 tariffs at any hour. In the screenshot below, the timeslot consulted is 8h00 am.



It can be observed that the default tariff has an almost constant price independently of the time of the day. For this reason, it is not an interesting tariff for the user to achieve savings using demand response methods. For this purpose, it is more interesting to contract the 2.0 DHA or the 2.0 DHS tariff, where the price varies depending on the hour. However, it can be interesting to analyse your project load consumption using the 3 tariffs to compare which one is the most economical in function of the load profile.

API documentation

At this point, we have already worked with a simple REST API and treated the obtained JSON. The next step will be to use this acquired knowledge to work with the desired API for our personal project. In this case, the REE API.

The documentation of this API is accessible using the following link:

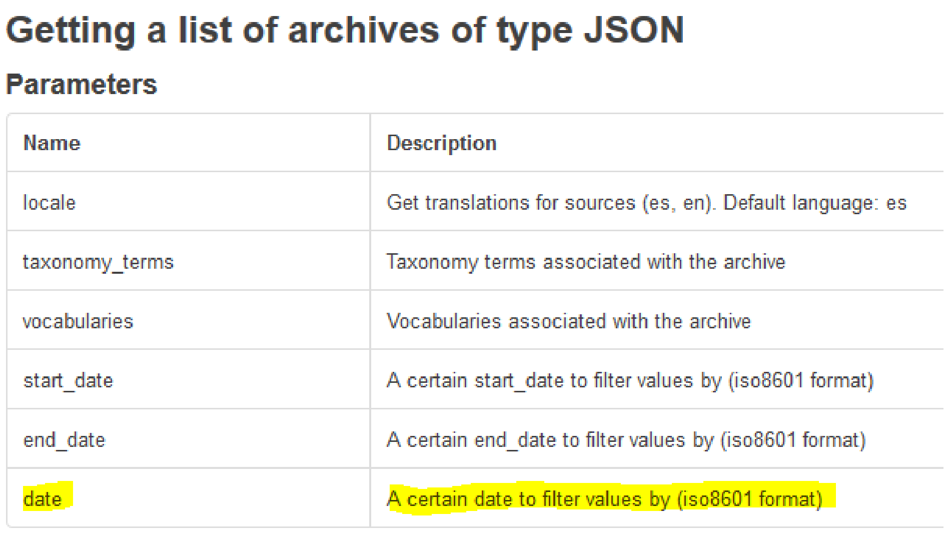
https://api.esios.ree.es/

The documentation available is too extensive, and it is not straightforward to understand. However, it is not necessary to use all the available functions of the API. Whether you enter the main web page of the documentation, you will find a subsection dedicated on how to operate with JSON archives.

In this subsection, we have four available options. It is possible to access a specific JSON archive. However, at this point, we don't have the information about which archive we need. So the first step will be to get a list of all the JSON available archives using the option:

Getting a list of archives of type JSON

Once the document is opened, we have a list of the parameters that can be used with the API and examples of how to construct the header to make the GET request.

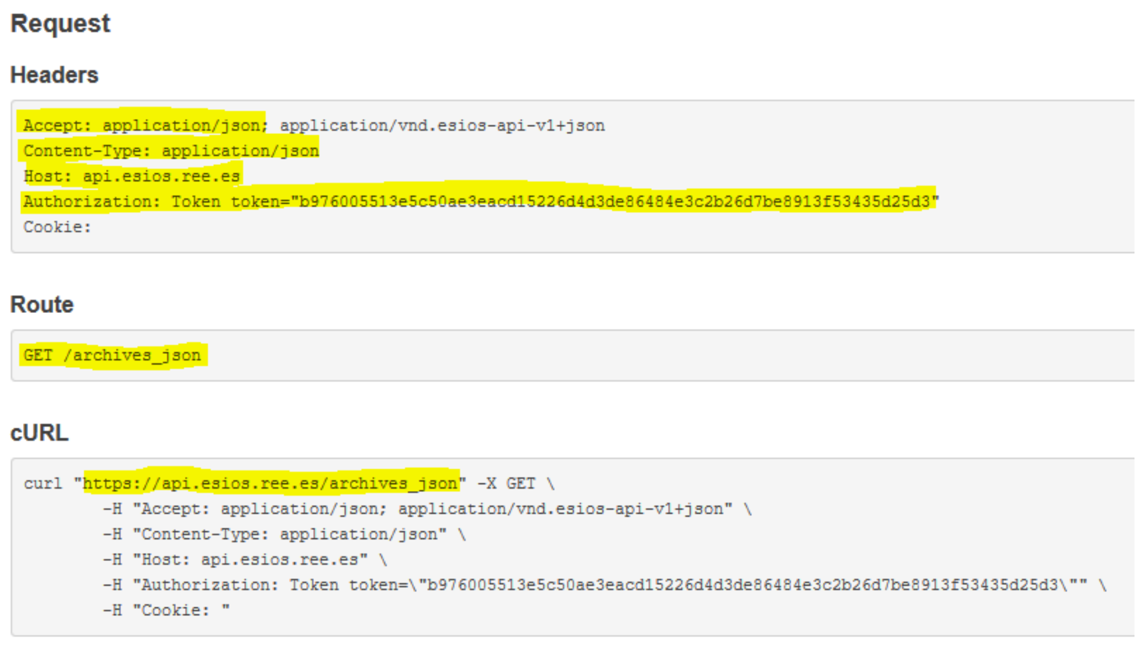


Getting a list of archives of type JSON

Parameters

Name	Description
locale	Get translations for sources (es, en). Default language: es
taxonomy_terms	Taxonomy terms associated with the archive
vocabularies	Vocabularies associated with the archive
start_date	A certain start_date to filter values by (iso8601 format)
end_date	A certain end_date to filter values by (iso8601 format)
date	A certain date to filter values by (iso8601 format)

For our project, the only parameter we need is the *date* to obtain the price of the energy of the present day or for the day ahead. For this purpose, it is not necessary to use the *start_date* and *end_date* parameters, more useful to consult past information.


Request

Headers

```
Accept: application/json; application/vnd.esios-api-v1+json
Content-Type: application/json
Host: api.esios.ree.es
Authorization: Token token="b976005513e5c50ae3eacd15226d4d3de86484e3c2b26d7be8913f53435d25d3"
Cookie:
```

Route

```
GET /archives_json
```

cURL

```
curl "https://api.esios.ree.es/archives_json" -X GET \
        -H "Accept: application/json; application/vnd.esios-api-v1+json" \
        -H "Content-Type: application/json" \
        -H "Host: api.esios.ree.es" \
        -H "Authorization: Token token=\"b976005513e5c50ae3eacd15226d4d3de86484e3c2b26d7be8913f53435d25d3\"" \
        -H "Cookie: "
```

On the contrary to the Bicing API example, the REE API needs a more complex header to realize the GET request. So it is necessary to generate a correct header with the highlighted information, where the token will be the personal token given to you by REE. If you use the default token in the screenshot, the request will fail.

REE API script

Now is your turn to write the code of the script. This process will be guided by this document. For any doubt, ask your professor for further assistance.

Step 1. Creation of the script

Using a Python IDE such as Spyder, to create a new file called REE_API.py for example.

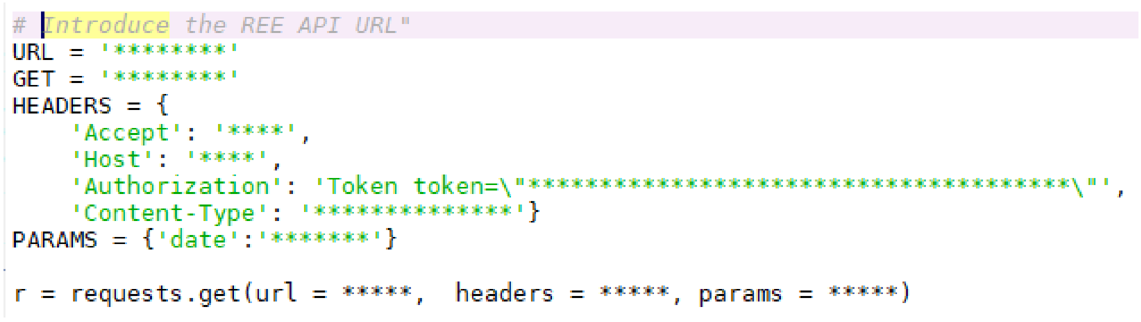
Step 2. Import the required libraries

As done in the Bicing API, we only need to work with the *requests* library

Step 3. Create the request data

To make a GET request in this API, we need to add an *url*, and also a *headers* and at least one *params*.

The *url* is better if it is divided into two different parts (URL + GET) as explained in the Bicing example. It will be useful at the end of the script. All the information can be extracted from the previous screenshot except your personal token.



```
# Introduce the REE API URL"
URL = '*********'
GET = '*********'
HEADERS = {
     'Accept': '****',
     'Host': '****',
     'Authorization': 'Token token=\"***************************************\"',
     'Content-Type': '***************'}
PARAMS = {'date':'*******'}

r = requests.get(url = *****,  headers = *****, params = *****)
```

Once the URL, GET, HEADERS, and PARARMS variables are filled with the correct values, do a GET request using these variables. You have to substitute all the **** with the correct information.

Step 4. Check the server answer

- Step 4.1. First, check the status code. You can print this value using:

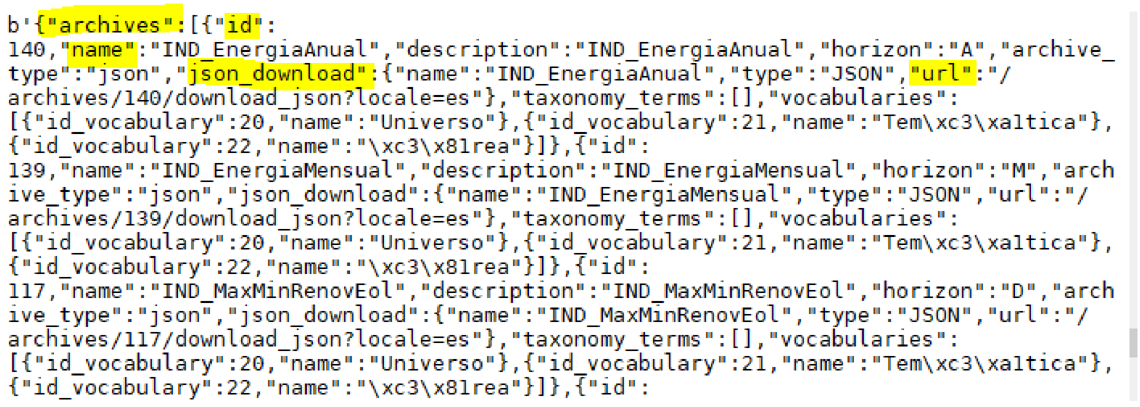
```
Print(r.status_code)
```

If the value is a 200, it means that the GET petition has been done properly.

- Step 4.2. If the answer is ok, it means we have stored in the *r* variable a JSON archive. But we don't know the structure of this JSON. First, print the content of the *r* object.

```
Print(r.content)
```

After printing the content, you will obtain something like this:



```
b'{"archives":[{"id":
140,"name":"IND_EnergiaAnual","description":"IND_EnergiaAnual","horizon":"A","archive_
type":"json","json_download":{"name":"IND_EnergiaAnual","type":"JSON","url":"/
archives/140/download_json?locale=es"},"taxonomy_terms":[],"vocabularies":
[{"id_vocabulary":20,"name":"Universo"},{"id_vocabulary":21,"name":"Tem\xc3\xa1tica"},
{"id_vocabulary":22,"name":"\xc3\x81rea"}]},{"id":
139,"name":"IND_EnergiaMensual","description":"IND_EnergiaMensual","horizon":"M","arch
ive_type":"json","json_download":{"name":"IND_EnergiaMensual","type":"JSON","url":"/
archives/139/download_json?locale=es"},"taxonomy_terms":[],"vocabularies":
[{"id_vocabulary":20,"name":"Universo"},{"id_vocabulary":21,"name":"Tem\xc3\xa1tica"},
{"id_vocabulary":22,"name":"\xc3\x81rea"}]},{"id":
117,"name":"IND_MaxMinRenovEol","description":"IND_MaxMinRenovEol","horizon":"D","arch
ive_type":"json","json_download":{"name":"IND_MaxMinRenovEol","type":"JSON","url":"/
archives/117/download_json?locale=es"},"taxonomy_terms":[],"vocabularies":
[{"id_vocabulary":20,"name":"Universo"},{"id_vocabulary":21,"name":"Tem\xc3\xa1tica"},
{"id_vocabulary":22,"name":"\xc3\x81rea"}]},{"id":
```

Step 5. Analyse the JSON file.

The previous command prints the JSON file stored in the *r* variable. Be careful because the main JSON field is not 'stations' as the Bicing API example.

It is important to analyse that this JSON returns a list of different accessible archives. Each of these archives has a parameter 'id,' 'name' and 'JSON_download/url field. The most interesting fields are highlighted.

Write a *for* command that prints the 'id' and 'name' field. You have to achieve a list printed as the following one:

```
140 - IND_EnergiaAnual
139 - IND_EnergiaMensual
117 - IND_MaxMinRenovEol
116 - IND_MaxMin
115 - IND_DemandaRealGen
114 - IND_DemandaPrevProg
113 - EntitledParticipants
112 - BalanceResponsibleParties
111 - ProgrammingUnits
110 - GenerationUnits
84 - ParticipantesSubasta
83 - SujetosMercado
82 - UnidadesProgramacion
81 - UnidadesFisicas
77 - PVPC_VHC_P3_DD
76 - PVPC_VHC_P2_DD
75 - PVPC_VHC_P1_DD
74 - PVPC_NOC_P2_DD
73 - PVPC_NOC_P1_DD
72 - PVPC_GEN_P1_DD
70 - PVPC_CURV_DD
67 - IND_Umbrales
66 - IND_PrecioFinal
65 - IND_PrecioDesvios
64 - IND_PotenciaInstalada
63 - IND_Interconexiones
62 - IND_DemandaInterrumpible
```

Step 6. Obtain the associated URL to the desired ID indicator

Depending on the indicator we are interested in, we can do a new GET petition with the URL given by the JSON within the 'JSON_download/url' field.

With the next piece of code, the user can write as an input with the keyboard the desired indicator to consult.

```
DATA_ID = int(input("Please enter the data id: "))

for arch in r.json()['*****']:
        if arch['id'] is DATA_ID:
                GET = arch['******']['***']

print ('Getting ' + str(DATA_ID) + ':' + GET)
```

After storing the desired ID in the DATA_ID integer variable, we can implement a *for* command to look for the 'JSON_download/url of the desired ID.

Substitute the ***** characters with the correct information. And with the print command, show to the screen the desired URL resource.

Step 7. Make a new GET request

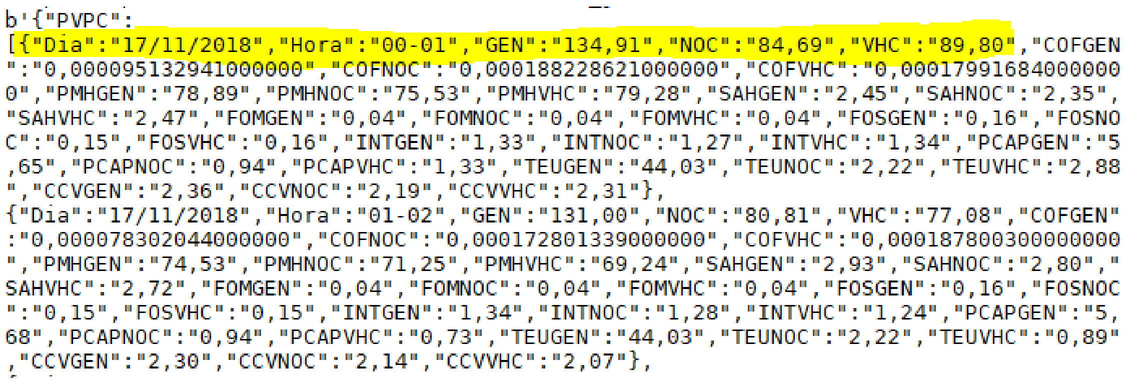
We have rewritten the GET variable with the new resource we want to access. So now we can make a new request. The *headers* and the *params* are the same as the first petition. So it is not needed to write the HEADER and PARAMS variables again.

```
r = requests.get(url = URL+GET, headers = *****, params = ******)

print( r.status_code)
print( r.url)
print( r.content)
```

Print the new JSON to check the answer of the server.

If your print the indicator 70 for example, associated to the PVPC tariffs, the printed information will look like this:



```
b'{"PVPC":
[{"Dia":"17/11/2018","Hora":"00-01","GEN":"134,91","NOC":"84,69","VHC":"89,80","COFGEN
":"0,000095132941000000","COFNOC":"0,000188228621000000","COFVHC":"0,00017991684000000
0","PMHGEN":"78,89","PMHNOC":"75,53","PMHVHC":"79,28","SAHGEN":"2,45","SAHNOC":"2,35",
"SAHVHC":"2,47","FOMGEN":"0,04","FOMNOC":"0,04","FOMVHC":"0,04","FOSGEN":"0,16","FOSNO
C":"0,15","FOSVHC":"0,16","INTGEN":"1,33","INTNOC":"1,27","INTVHC":"1,34","PCAPGEN":"5
,65","PCAPNOC":"0,94","PCAPVHC":"1,33","TEUGEN":"44,03","TEUNOC":"2,22","TEUVHC":"2,88
","CCVGEN":"2,36","CCVNOC":"2,19","CCVVHC":"2,31"},
{"Dia":"17/11/2018","Hora":"01-02","GEN":"131,00","NOC":"80,81","VHC":"77,08","COFGEN"
:"0,000078302044000000","COFNOC":"0,000172801339000000","COFVHC":"0,000187800300000000
","PMHGEN":"74,53","PMHNOC":"71,25","PMHVHC":"69,24","SAHGEN":"2,93","SAHNOC":"2,80","
SAHVHC":"2,72","FOMGEN":"0,04","FOMNOC":"0,04","FOMVHC":"0,04","FOSGEN":"0,16","FOSNOC
":"0,15","FOSVHC":"0,15","INTGEN":"1,34","INTNOC":"1,28","INTVHC":"1,24","PCAPGEN":"5,
68","PCAPNOC":"0,94","PCAPVHC":"0,73","TEUGEN":"44,03","TEUNOC":"2,22","TEUVHC":"0,89"
,"CCVGEN":"2,30","CCVNOC":"2,14","CCVVHC":"2,07"},
```

There are highlighted the most relevant values that we are interested:

- Dia: day
- Hora: hour
- GEN: price of the default tariff (2.0 A)
- NOC: price of the 2 periods tariff (2.0 DHA)
- VHC: price of the electric vehicle tariff (2.0 DHS)

The following values are coefficients and corrections applied to calculate the price of the tariffs.

On the contrary, if we print the information of the indicator 66, associated with the final prices, the information obtained is:

- GEN: price of the default tariff (2.0 A)
- NOC: price of the 2 periods tariff (2.0 DHA)
- VHC: price of the electric vehicle tariff (2.0 DHS)
- COF: Profile of the default tariff
- PMHGEN: Daily and intraday component of the default tariff.
- PMHNOC: Daily and intraday component of the 2 periods tariff.
- PMHVHC: Daily and intraday component of the electric vehicle tariff.
- SAHGEN: Adjustment service component of the default tariff.
- SAHNOC: Adjustment service component of the 2 periods tariff.
- SAHVHC: Adjustment service component of the electric vehicle tariff.
- BOEGEN: Regulated components at the BOE (Spanish Official State Gazette) of the default tariff.
- BOENOC: Regulated components at the BOE of the 2 periods tariff.
- BOEVHC: Regulated components at the BOE of the electric vehicle tariff.

If you have any doubts related to the meaning of these parameters or any other ones of other indicators, you can send an e-mail to consultasios@ree.es, to ask for clarifications about the API.

Step 8. Extract the desired information

At this point, we have printed the whole JSON data, but usually, we will only want to extract 2 values: the price of the energy of the 2-period tariff (for example) and the correspondent hour. Hence, write a *for* command over the JSON file and print only the time of the day and the price of the NOC (2 periods) tariff.

```
for arch in r.json()['PVPC']:
        print('Price at ' + str(arch['Hora']) + 'h is : ' + str(arch['NOC']) + ' €/MWh')
```

The result is something like this.

```
Price at 00-01h is : 84,69 €/MWh
Price at 01-02h is : 80,81 €/MWh
Price at 02-03h is : 75,71 €/MWh
Price at 03-04h is : 73,16 €/MWh
Price at 04-05h is : 69,37 €/MWh
Price at 05-06h is : 75,33 €/MWh
Price at 06-07h is : 74,78 €/MWh
Price at 07-08h is : 77,84 €/MWh
Price at 08-09h is : 79,67 €/MWh
Price at 09-10h is : 83,71 €/MWh
Price at 10-11h is : 82,95 €/MWh
Price at 11-12h is : 78,53 €/MWh
Price at 12-13h is : 143,00 €/MWh
Price at 13-14h is : 143,80 €/MWh
Price at 14-15h is : 145,33 €/MWh
Price at 15-16h is : 143,80 €/MWh
Price at 16-17h is : 146,26 €/MWh
Price at 17-18h is : 151,38 €/MWh
Price at 18-19h is : 155,57 €/MWh
Price at 19-20h is : 155,96 €/MWh
Price at 20-21h is : 150,44 €/MWh
Price at 21-22h is : 149,73 €/MWh
Price at 22-23h is : 78,08 €/MWh
Price at 23-24h is : 75,97 €/MWh
```

Remember that for your personal project you will want to save an array with only the price of the desired tariff.