**SESTO MAGNUS**

**AMR + PYTHON**



This document aims to explain the use of Python to control the SESTO Magnus AMR.

|  |  |
| --- | --- |
| Document revision | 1.0 |
| Document release date | 5 April 2021 |
| Remarks | New Release |

# Table of Contents

[Table of Contents 2](#_Toc68507902)

[1. General Information 3](#_Toc68507903)

[1.1 Introduction 3](#_Toc68507904)

[1.2 Requirements 3](#_Toc68507905)

[1.3 Definitions 3](#_Toc68507906)

[2. Getting Started 4](#_Toc68507907)

[2.1 Installation 4](#_Toc68507908)

[2.2 How SESTO’s REST API works 6](#_Toc68507909)

[2.3 Application of SESTO’s REST API 7](#_Toc68507910)

[3. AMR\_ELEMENTS Python Module 11](#_Toc68507911)

[3.1 Class AMRServerElement 11](#_Toc68507912)

[3.2 Class AMRElement 13](#_Toc68507913)

[3.3 AMR\_ELEMENTS examples 15](#_Toc68507914)

# General Information

## 1.1 Introduction

The SESTO Magnus AMR is built-in with its SESTO REST API also known as RESTful API which is used to communicate with the AMR. An example of how this API is used is when the user sends a goal for the AMR.

## 1.2 Requirements

Python Integrated Development Environment (IDE) and a Python 3 interpreter will be required. In this document, PyCharm will be used as the IDE. The “Requests” python module will also be required.

## 1.3 Definitions

**REST API:** Representational State Transfer (REST) is a software structure that uses a subset of HTTP. It is also known as RESTful API and both can be used interchangeably.

**GET:** Method used to request information from the web service.

**POST:** Method used to post data to a web server, like the PUT method however it does not replace duplicates, for example sending the same goal, multiple times.

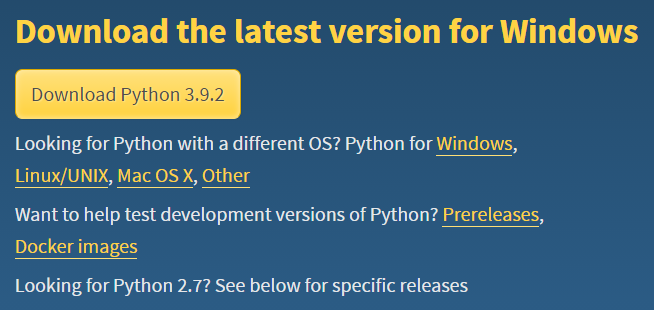
**PUT:** Method used to put data to a web server, like the POST method however it will update resources that already exists, for example putting the AMR to Pause, it will just update the AMR pause resource to “paused” from “operational”.

**DELETE:** Method is used to delete data from a web server.

# Getting Started

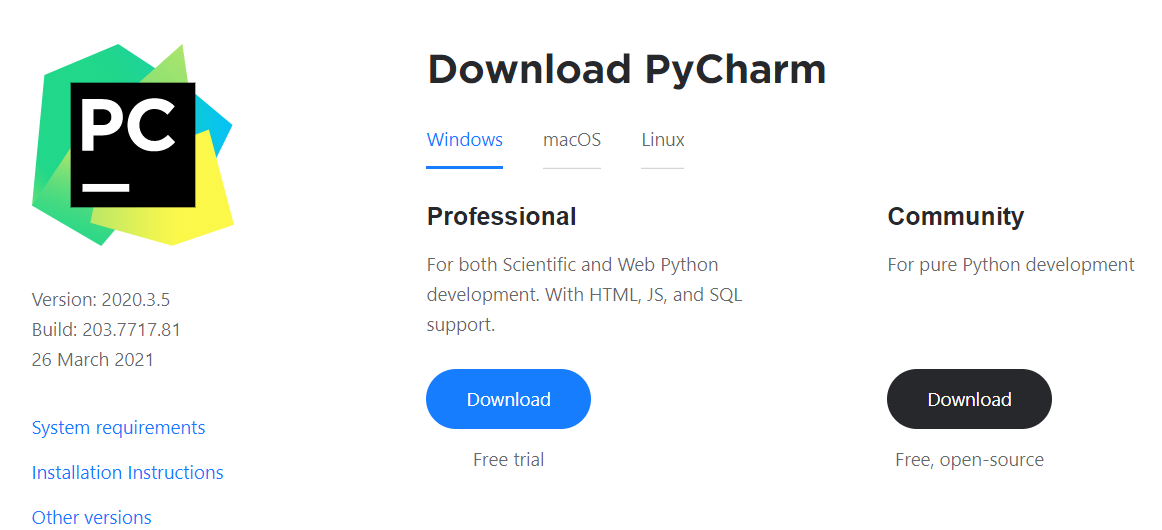
## 2.1 Installation

Firstly, you will need to have Python 3 installed. To do so go to <https://www.python.org/downloads/> and then download the latest version of Python 3.

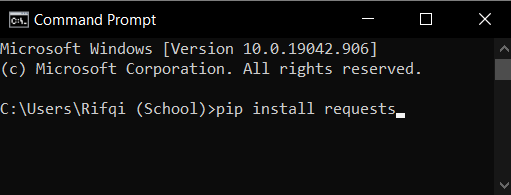


Select the right operating system based on your PC. You may refer to this website for a guide on how to install Python 3: <https://phoenixnap.com/kb/how-to-install-python-3-windows>.

Secondly, you will need to have a Python IDE installed, in this case we will be using PyCharm. To do so go to <https://www.jetbrains.com/pycharm/download/>. You can click on installation instructions and go to <https://www.jetbrains.com/help/pycharm/installation-guide.html> for more detail on how to install PyCharm.



Lastly, you will need to install the “Requests” python module it may be installed automatically when you use the code “import requests” in the IDE, or you may install it using Windows built-in command prompt or Terminal if you are using Mac or Linux, and to do so type in “pip install requests” as shown in the example below:



## 2.2 How SESTO’s REST API works

Like other REST APIs, SESTO’s REST API communicate with the AMR through GET, POST, PUT, and DELETE which are HTTP request methods.

Firstly, the GET method is used to request information from the web service.

Secondly, the POST method is used to post data to a web server, like the PUT method however it does not replace duplicates, for example sending the same goal, multiple times.

Thirdly, the PUT method is used to put data to a web server, like the POST method however it will update resources that already exists, for example putting the AMR to Pause, it will just update the AMR pause resource to “paused” from “operational”.

Lastly, the DELETE method is used to delete data from a web server.

These request methods help with the communication by sending and receiving data as a means of communication between client and server, for instance, from PC to Server to AMR, and vice-versa.

## 2.3 Application of SESTO’s REST API

There are many ways to use the SESTO’s REST API such as using the provided Fleet UI to do specific actions. In this document, Python scripts will be used to explain how the SESTO’s REST API can be used. You will need to refer to SESTO’s Element manuals.

Firstly, make sure to import “requests” and “json” to the script. The requests module will be used to send and receive data from the website and the JSON module will be used to convert Python variables to string, to be used when sending data to the AMR or the Fleet Server.

import requests  
import json

Secondly, we create two classes to make it simpler to use in the future, AMRServerElement and AMRElement. AMR Server Element will be used for communicating from your PC to the Fleet Server and AMR Element will be used for communicating from your PC to the AMR. Both classes will be similar but will have slight differences in how it is used.

class AMRServerElement

class AMRElement

Inside AMRServerElement class, we create a function \_\_init\_\_ which is a constructor for the class. In this Class, we require the IP of the fleet server, and the login credentials of the fleet server.

def \_\_init\_\_(self, ip="192.168.0.249", username="admin", password="P@ssword123"):  
 self.username = username  
 self.password = password  
 self.ip = ip  
 headers = {  
 "user\_name": "{}".format(username), # insert username  
 "password": "{}".format(password) # insert password  
 }  
 url = 'http://' + ip + "/public/auth"  
 if not self.pingable():  
 self.pingable\_result = False  
 return  
 r = requests.post(url, None, headers)  
 if r.status\_code >= 400:  
 self.token = "invalid"  
 self.authorization = "invalid"  
 return  
 self.token = r.json()['access\_token']  
 self.authorization = {  
 'Authorization': 'Bearer {}'.format(self.token),  
 'Content-Type': 'application/json',  
 "Cache-Control": "no-cache"  
 }

What these lines of code does is, it uses POST to get the authentication token. Previously, it was mentioned that POST is used to send data but how do we get the authentication token? It is done by getting the JSON passed from the request made. Tokens are used for authentication so that you will be able to access and send more data to the server.

In this case, we post data by typing assigning variable r = request.post() and passing it the HTTP address, which is the variable url, and passing no data to it thus None and passing to the headers the headers dictionary which contains the login credentials used to login to the SESTO fleet server.

Afterwards we do an error checking by checking the status code, if the status code is not >= 400 it means the login credentials are correct and that you can get the token. We assign token by using r.json() which will get back the JSON value passed after doing the request, and we only want the access\_token value thus we type [‘access\_token’] beside it.

We then create another dictionary, authorization which contains the important values when doing requests to the server, an analogy will be that it is like the pages after the login page, and that is where the token is used. The ‘Authorization’ value will be used for the authentication, and the ‘Content-Type’ will be used to get back json as the value after the request, for ‘Cache-Control’ it is so that nothing is cached.

There is also the pingable function used which is used to prevent errors when the AMR or the fleet server is not reachable or contactable due to network issues.

def pingable(self):  
 try:  
 requests.get('http://' + self.ip)  
 except Exception as e:  
 print(e)  
 self.pingable\_result = False  
 return False  
 else:  
 self.pingable\_result = True  
 return True

Inside AMRElement class, we create a function \_\_init\_\_ which is a constructor for the class. In this Class, we only require the IP of the AMR.

def \_\_init\_\_(self, ip="192.168.1.248"):  
 self.ip = ip  
 if not self.pingable():  
 self.pingable\_result = False  
 return  
 self.headers = {  
 "Content-Type": "application/json",  
 "Cache-Control": "no-cache",  
 }

We only check if the AMR can be reached or contacted and then we create the headers variable which will be used when doing requests.

The headers dictionary used in this class is like the authorization dictionary used in AMRServerElement class, except it does not require any authentication.

Lastly, we created a few functions inside both the classes, get\_json(), post\_json() and put\_json(). They are similar except the headers and the HTTP address used whereas the AMRServerElement class uses self.authorization and its url has /public after the IP address and the AMRElement class uses self.headers and its url has /amr/public after the IP address.

**AMRServerElement**

def get\_json(self, url\_append):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/public'  
 url = url + url\_append  
 r = requests.get(url, headers=self.authorization)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()  
  
def post\_json(self, url\_append, data):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/public'  
 url = url + url\_append  
 r = requests.post(url, json.dumps(data), headers=self.authorization)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()  
  
def put\_json(self, url\_append, data):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/public'  
 url = url + url\_append  
 r = requests.put(url, json.dumps(data), headers=self.authorization)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()

**AMRElement**

def get\_json(self, url\_append):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/amr/public'  
 url = url + url\_append  
 r = requests.get(url)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()  
  
def post\_json(self, url\_append, data):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/amr/public'  
 url = url + url\_append  
 r = requests.post(url, json.dumps(data), headers=self.headers)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()  
  
def put\_json(self, url\_append, data):  
 if not self.pingable\_result:  
 return  
 url = 'http://' + self.ip + '/amr/public'  
 url = url + url\_append  
 r = requests.put(url, json.dumps(data), headers=self.headers)  
 print('Status Code: {}'.format(r.status\_code))  
 try:  
 print("JSON from {}:".format(url))  
 print(r.json())  
 except Exception as e:  
 print(e)  
 return False  
 else:  
 return r.json()

The url\_append variable is used to add on to the url such as ‘/goals’ for example, and the data variable is used to pass data during the requests.

You can learn more about the ‘requests’ module online.

# AMR\_ELEMENTS Python Module

In 2.3, we have shown some examples and explained how we can use SESTO’s REST API. To make it easier, we have made our own amr\_elements.py module that can be used to communicate with the fleet server and the AMR. We have also made more functions to make it easier to use the module instead of using some of the functions we have shown in 2.3. You may get our updated version here: <https://github.com/RifqiHarits/AMR_Project>. To use the module just import amr\_elements in your script.

## 3.1 Class AMRServerElement

Below is a list of functions you may use for the AMRServerElement class.

|  |  |
| --- | --- |
| AMRServerElement Functions | Description |
| AMRServerElement(ip, username, password) | This is used to create an object of this class. The default ip is 192.168.0.249, the default username is admin, and the default password is P@ssword123. |
| pingable() | To check whether you can access the fleet server by sending requests to the IP of the Server. |
| request\_token() | To get the authentication token that is required for sending requests that requires authentication. |
| get\_json(url\_append) | To GET JSON from the extended URL for example (‘/goals’) means <http://serverip/public/goals> . |
| post\_json(url\_append, data) | To POST JSON to the extended URL for example (‘/goals’, data) means <http://serverip/public/goals> . The POST data come from data in the form of dictionary. |
| put\_json(url\_append, data) | To PUT JSON to the extended URL for example (‘/goals’, data) means <http://serverip/public/goals> . The POST data come from data in the form of dictionary. |
| get\_pauses() | To get AMR(s) pause status information. |
| pause\_amrs(amr\_ids) | To pause AMR(s) by sending a list, for example amr\_ids = [0,1,2]. |
| resume\_amrs(amr\_ids) | To resume AMR(s) by sending a list, for example amr\_ids = [0,1,2]. |
| get\_modes() | To get the AMR(s) mode status information. |
| get\_idle\_release\_statuses() | To get AMR(s) idle release status information. |
| get\_station\_release\_statuses() | To get AMR(s) station release status information. |
| get\_users() | To get user(s) registered to the fleet server. |
| get\_version() | To get version of the fleet server. |
| get\_graphs() | To get graph ids. |
| get\_maps() | To get map ids. |
| send\_goal(amr\_id, graph\_id, waypoint\_id) | To send goal to the chosen AMR based on the amr\_id, graph\_id, and waypoint\_id. |
| station\_release(amr\_id) | To release the chosen AMR from the station based on the amr\_id. |
| idle\_release(amr\_id) | To send the chosen AMR to parking based on the amr\_id. |
| get\_waypoint(amr\_id) | To get waypoint of the chosen AMR based on the amr\_id. |
| wait\_waypoint\_reached(amr\_id, waypoint\_id) | To wait until the chosen AMR have reached its waypoint based on the amr\_id and waypoint\_id. |
| send\_goal\_auto\_release(amr\_id, graph\_id, waypoint\_id) | This function is a combination of send\_goal(), wait\_waypoint\_reached() and station\_release(). |
| send\_idle\_wait\_released(amr\_id) | This function is a combination of idle\_release(), wait\_waypoint\_reached() and an addition of a few codes in the function. |
| create\_task(data) | This function creates task with the data (follow data format based on the API manual). |
| get\_tasks() | This function retrieves the current tasks in the AMR Fleet Server. |
| update\_task(data) | This function updates task with the data (follow data format based on the API manual). |
| cancel\_task(task\_id) | This function cancels task based on the task id. |
| pause\_task(task\_id) | This function pauses task based on the task id. |
| resume\_task(task\_id) | This function resumes task based on the task id. |

## 3.2 Class AMRElement

Below is a list of functions you may use for the AMRElement class.

|  |  |
| --- | --- |
| AMRElement Functions | Description |
| AMRElement(ip) | This is used to create an object of this class. The default ip is 192.168.1.248. |
| pingable() | To check whether you can access the fleet server by sending requests to the IP of the AMR. |
| request\_token() | To get the authentication token that is required for sending requests that requires authentication. |
| get\_json(url\_append) | To GET JSON from the extended URL for example (‘/controls/pause’) means <http://serverip/amr/public/controls/pause> . |
| post\_json(url\_append, data) | To POST JSON to the extended URL for example (‘/controls/pause’, data) means <http://serverip/amr/public/controls/pause> . The POST data come from data in the form of dictionary. |
| put\_json(url\_append, data) | To PUT JSON to the extended URL for example (‘/controls/pause’, data) means <http://serverip/amr/public/controls/pause> . The POST data come from data in the form of dictionary. |
| get\_released() | Returns True when the AMR is released. |
| wait\_released() | Wait until the AMR is released by checking get\_released() return value. |
| get\_paused() | Returns True when the AMR is paused. |
| pause\_amr() | Pauses the AMR. |
| resume\_amr() | Resume the AMR from “paused” state. |
| get\_battery\_status() | Gets the battery status information. |
| get\_main\_battery\_percentage() | Gets the main battery percentage. |
| get\_amr\_id() | Gets the id of the AMR. |
| get\_amr\_name() | Gets the name of the AMR. |
| get\_input\_pin(pin) | Get the Boolean value of input pin based on variable pin. |
| get\_input\_pin\_all() | Get the Boolean value of all input pins. |
| get\_output\_pin(pin) | Get the Boolean value of output pin based on variable pin. |
| get\_output\_pin\_all() | Get the Boolean value of all output pins. |

## 3.3 AMR\_ELEMENTS examples

### 3.3.1 Sending goals manually

import amr\_elements  
import threading  
  
# AMC GRAPH #  
graph\_id = 'cd0f463f-17f5-4f35-aa7e-a99c5c28bdfd'  
  
amr = amr\_elements.AMRServerElement("192.168.0.249")  
amr\_6 = amr\_elements.AMRElement("192.168.1.248")  
amr\_7 = amr\_elements.AMRElement("192.168.1.249")  
  
  
def main():  
 # Set True to respective AMR if they will be used #  
 amr6\_use = True  
 amr7\_use = True  
  
 charging\_waypoint = 96  
  
 def choice(amr\_id):  
 ans = input("Do you want to continue task for AMR {}? (Y/N): ".format(amr\_id))  
 if ans.lower() == "y":  
 return True  
 else:  
 return False  
  
 # This will loop if AMR Server is offline #  
 while not amr.pingable\_result:  
 amr.pingable()  
 amr.request\_token()  
  
 # Get AMR Statuses #  
 amr.get\_json('/amrs/statuses')  
  
 # Codes for AMR 6 to be written in the while loop inside function amr6() #  
 def amr6():  
 if not amr6\_use:  
 return  
 # This will loop if AMR 6 is offline #  
 while not amr\_6.pingable\_result:  
 amr\_6.pingable()  
  
 amr\_id = 6  
 # Making sure AMR 6 starts from RC1 #  
 if amr.get\_waypoint(amr\_id) != 1:  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 1)  
  
 # This will let it loop forever if AMR 6 remains online #  
 while amr\_6.pingable():  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 4)  
 amr.wait(4)  
 amr.send\_idle\_wait\_reached(amr\_id)  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 3)  
 amr.wait(4)  
 if amr\_6.get\_main\_battery\_percentage() < 40:  
 break  
 # if not choice(amr\_id):  
 # break  
 # Send to RC1 #  
 if amr\_6.pingable():  
 if amr\_6.get\_main\_battery\_percentage() < 40:  
 amr.send\_goal(amr\_id, graph\_id, charging\_waypoint)  
 while amr\_6.get\_main\_battery\_percentage() < 95:  
 amr.wait(120)  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 1)  
  
 # Codes for AMR 7 to be written in the while loop inside function amr7() #  
 def amr7():  
 if not amr7\_use:  
 return  
 # This will loop if AMR 7 is offline #  
 while not amr\_7.pingable\_result:  
 amr\_7.pingable()  
  
 amr\_id = 7  
 # Making sure AMR 7 starts from RC46 #  
 if amr.get\_waypoint(amr\_id) != 46:  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 46)  
  
 # This will let it loop forever if AMR 7 remains online #  
 while amr\_7.pingable():  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 3)  
 amr.wait(4)  
 amr.send\_idle\_wait\_reached(amr\_id)  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 4)  
 amr.wait(4)  
 if amr\_7.get\_main\_battery\_percentage() < 40:  
 break  
 # if not choice(amr\_id):  
 # break  
 # Send to RC46 #  
 if amr\_7.pingable():  
 if amr\_7.get\_main\_battery\_percentage() < 40:  
 amr.send\_goal(amr\_id, graph\_id, charging\_waypoint)  
 while amr\_7.get\_main\_battery\_percentage() < 95:  
 amr.wait(120)  
 amr.send\_goal\_auto\_release(amr\_id, graph\_id, 46)  
  
 # The codes below are used for concurrency and parallelism #  
 amr6\_thread = threading.Thread(target=amr6)  
 amr7\_thread = threading.Thread(target=amr7)  
  
 amr6\_thread.start()  
 amr7\_thread.start()  
  
  
main()