

Wago Cloud API with Python

Ex: Import/Export of an Alarm from one account to another

Disclaimer:

This documentation and code are not indicative of best practices in terms of HTTP handling and security. Proceed with the notion that there may very well be additional security measures necessary for your specific use case. This paper was written by a conceited Wago engineer that thinks of himself as a programmer and has a fascination with automated cloud processes.

Much like the Temple of Doom (a REALLY old movie starring Kylo Ren's dad), proceed at your own risk!

Intro:

Wago Cloud is an industrialized alternative to the many existing cloud solutions in the current Cloud ecosystem (AWS, Azure, Google Cloud, etc.). With Wago's specialized approach to the industrial automation world, we create a more streamlined progression while focusing on metric analysis, data management, dashboarding, and alarming.

While the features for the Wago Cloud are expansive and lend themselves to easy startup, Wago has integrated a REST-API for further scalability and automated management. The REST-API allows the user to gain access to many Wago Cloud data points such as subscriptions, alarm settings, live PLC data, and much more.

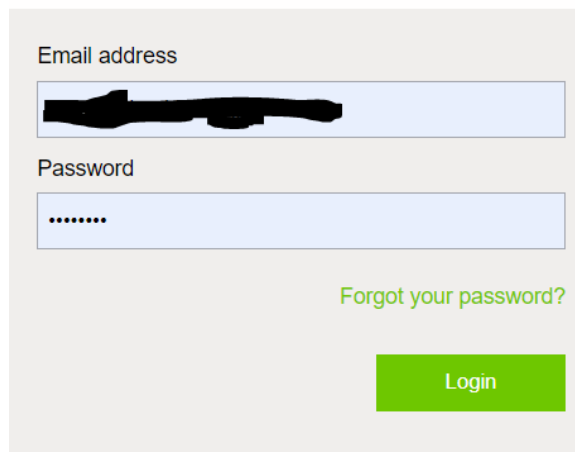
Ok, so now that I've earned my paycheck talking about how great a Wago product is, let's get into it!

Grabbing API key from Wago Cloud:

First things first, the API key is an authentication tool for utilizing the API. One of several layers of security for a secure cloud connection.

Let's go ahead and log into your Wago Cloud account via <https://cloud.wago.com>:

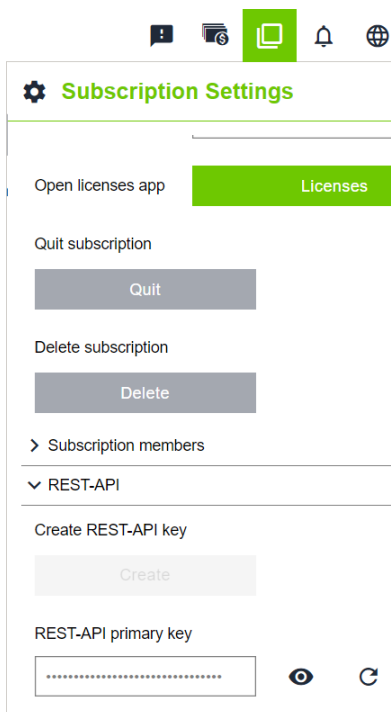
Login



The login form consists of two input fields: 'Email address' and 'Password'. The 'Email address' field contains a redacted email address. The 'Password' field contains a redacted password represented by seven dots. Below the password field is a green link that says 'Forgot your password?'. At the bottom right of the form is a green 'Login' button.

Figure 1: cloud.wago.com credentials

On the very top right of the screen, you'll click on the subscription settings button represented as a stack of pages shown in the picture below. Then you will scroll down to the REST-API tab and click "Create REST-API key":



The 'Subscription Settings' page has a sidebar with a gear icon and the title 'Subscription Settings'. The main content area includes several sections: 'Open licenses app' with a green 'Licenses' button; 'Quit subscription' with a grey 'Quit' button; 'Delete subscription' with a grey 'Delete' button; 'Subscription members' with a right-pointing arrow; 'REST-API' with a downward-pointing arrow; 'Create REST-API key' with a grey 'Create' button; and 'REST-API primary key' with a text input field containing redacted text, an eye icon, and a refresh icon.

Figure 2: Subscription settings in Wago Cloud to access API key

Important! You'll need to keep track of the API Key as this will be needed for most of the HTTP requests later on. If you lose the key, you'll need to create a new one and retroactively update your program.

Explaining HTTP requests in Python:

For this example, we will be utilizing the Requests library for python which is detailed in the attached link: <https://docs.python-requests.org/en/latest/>

The main two requests used within the program are GET & POST.

In short, the GET requests main function is for viewing data or other endpoints of interest, while a POST request is used for writing to the endpoint. Essentially, the POST allows you input data and the GET allows you to just ask for data.

More information about each request type can be found here:

https://www.w3schools.com/tags/ref_httpmethods.asp

Now that we got over that hot potato explanation, let's get into some cloud characteristics and Code.

Credentials for Wago Cloud API:

Remember when I said earlier that the API-key was a variable needed to access the cloud API? Well there is another credential needed by the user to initiate GET/POST commands to the API known as the Authorization Token.

The token value expires after 1 hour from the initial post request. It is recommended that the token is saved via an independent variable and re-requested once the hour has expired. In the case of my program which runs not often enough for that to be a concern, I request a new token with each program run.

In order to request a token there is some formatting that needs to be adhered too, pictured below:

4.1 Authorization via REST-API

To get an authorization token you can send a HTTP POST request to <https://cloud.wago.com/api/token> with the content type "application/x-www-form-urlencoded" and the following parameters in the request body:

- grant_type=password
- username=<your-username>
- password=<your-password>

Example with curl:

```
curl --location --request POST 'https://cloud.wago.com/api/token' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'grant_type=password' \
--data-urlencode 'username=***' \
--data-urlencode 'password=***'
```

You will receive a response in the following format:

```
{
  "access_token": "eyJ0eXAiOiJKV1QiLCJhbG...",
  "name": "<your-username>",
  "token_type": "bearer",
  "expires_in": 3600
}
```

The token expires after 1 hour and you have to retrieve a new one afterwards.

Note: We recommend caching the token and using it for its full duration. If you request a new authorization token for each API request, throttling may occur.

Using the token in API requests:

For all requests to the WAGO Cloud REST API, you need to add an Authorization Header. The value of the header is constructed with the token_type and access_token values, separated by a whitespace, e.g.:

Figure 3: Wago Cloud API documentation highlighting the Token post formatting

```
Authorization: bearer eyJ0eXAiOiJKV1QiLCJhbG...
```

Figure 4: Access Token format

Figure 3 is utilizing cURL, but now let's see how that looks in python and how to validate that the request was acknowledged with no errors.

Token POST request via Python:

```
import requests

#Setting up credentials for Token Check

API_Key = 'blahblah'
username = 'blahblahblah'
password = 'blah'

payload = {
    'grant_type': 'password',
    'username': username,
    'password': password
}
```

- ➔ To use the Python HTTP request library for GET/POST functions later in the program
- ➔ The Username/Password are the same two credentials used when logging into the Wago Cloud website
- ➔ Stuffing the variables needed into a JSON format. The specification for grant type was mentioned in figure 3.

Figure 5: Setting HTTP request payload via Python

The POST method for requesting a token is comprised of three parts: URL, Headers, and data (Figure 6).

The URL shows the absolute web path for the specific information specified.

The headers...well they do a lot. But in this case, the header is used to define how the data is being presented to and from the client.

The data is the information sent to the web service.

```
#Receiving and saving token info
Find_Token = requests.post(url='https://cloud.wago.com/api/token',
                           headers={'Content-Type': 'application/x-www-form-urlencoded'},
                           data=payload)

print(Find_Token)
print(Find_Token.json())
```

Figure 6: HTTP POST command via Python for token information

Regarding the output of this function, let's look at the two print statements below the Find_Token declaration.

Print(Find_Token) will output the HTTP response from the API. This will show whether the message was sent successfully or not.

Print(Find_Token.json()) will format the data received by the Cloud into a python dictionary, a format nearly identical to JSON.

```
print(Find_Token)

print(Find_Token.json())

Auth_Token = Find_Token.json().get('access_token')
```

Figure 7: Print statements for HTTP response and JSON Formatting

```
{
  "access_token": "eyJ0eX...qE0w",
  "refresh_token": "None",
  "name": "WAGOCLOUD.FAE@gmail.com",
  "Id": "fde15b85-94aa-4996-9e48-80f332c923f5",
  "token_type": "bearer",
  "expires_in": 5400,
  "expiredAt": "2022-03-03T19:11:47.8557778Z",
  "CreateTime": "2022-03-03T17:41:47.8557778Z"
}
```

Figure 8: JSON output string of Find_Token.json()

- ➔ <Response [200]> indicates a successful receiving of data
- ➔ <Response [404]> Indicates an incorrect link
- ➔ The output of a POST within the python requests library can automatically formatted into JSON. This makes data collection significantly easier.
- ➔ Referencing the JSON formatted messages to the left, I've used the .get() method to create the Auth_Token variable which houses the access_token data (eyJ...etc.)
- ➔ The json string also provides more info such as the account name, bearer, and expiration time.

Swagger for Wago Cloud API:

Now that we have all of the pertinent credentials necessary for reading and writing to the API, let's talk about what data we can get from the API itself.

A great resource for looking at possible Wago Cloud API data is the Swagger UI. Swagger allows you to look through all of the folders available through the API and the GET/PUT/POST commands within each folder or directory.

The link for the Swagger UI is here: <https://cloud.wago.com/api/doc/index.html>

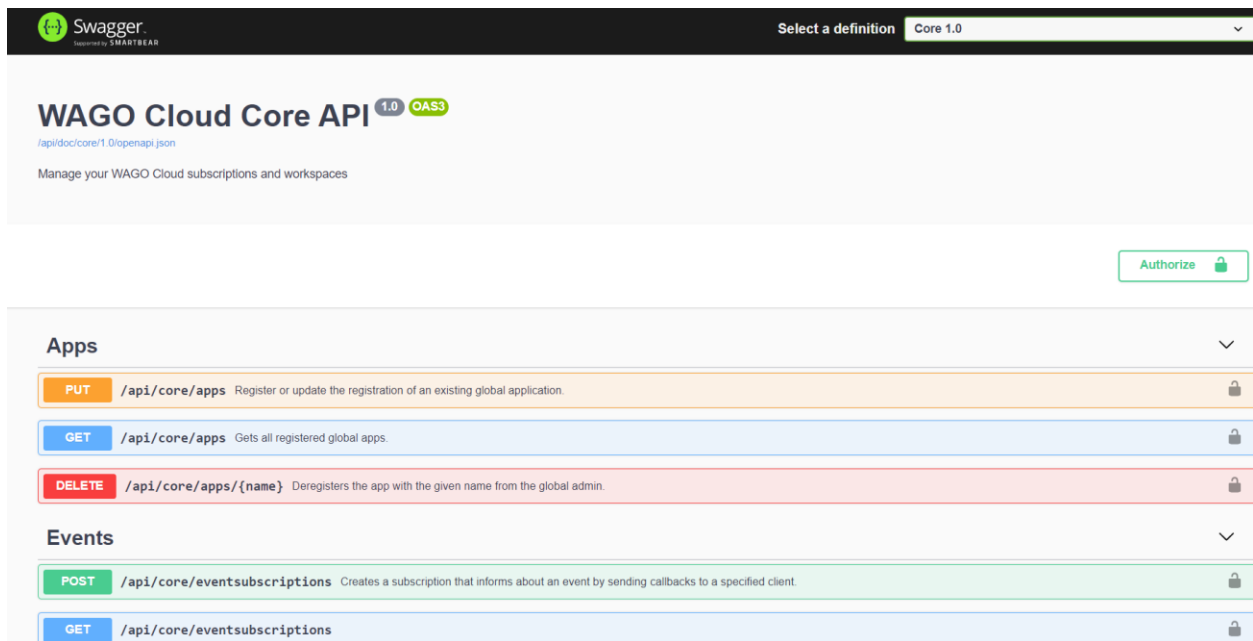


Figure 9: Swagger UI sample

By selecting the definition, you can view all HTTP commands within that directory.

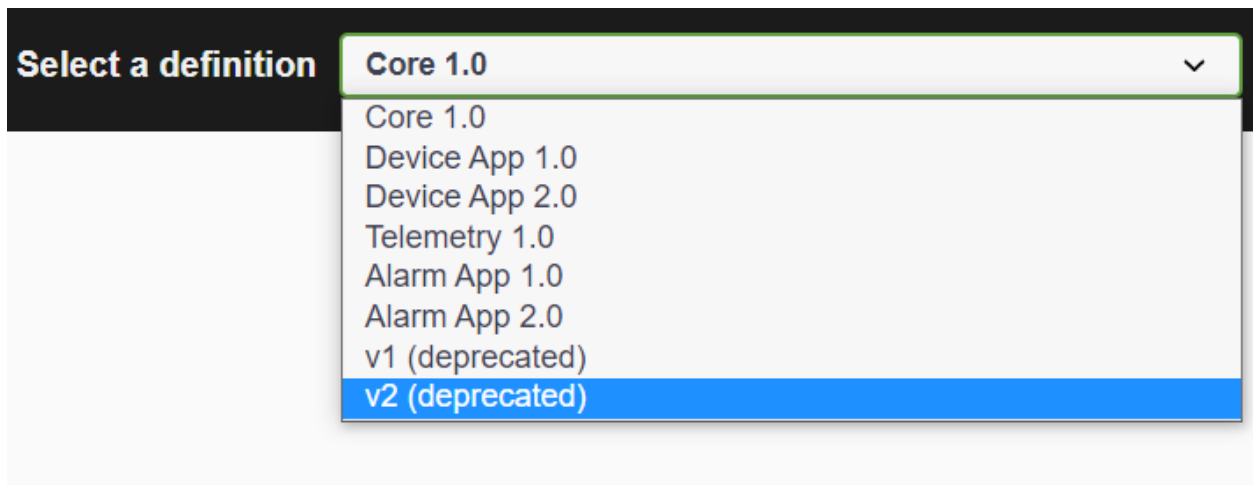


Figure 10: Swagger UI directory/definition drop down

In short, Swagger is a great way to test the function of your requests to the cloud and even get an idea of certain expected formats for each request.

Wago Cloud User Hierarchy:

Remember that the objective of this paper is to take an alarm from one cloud account to another account using this API.

Keeping that in mind, the first steps should be to find the account that contains the original alarm and the account that you'd like to copy that alarm to. The giving and receiving accounts, so to speak.

When finding the accounts, we'd need to understand how the accounts are structured. In Wago Cloud, the subscription houses all the users that the administrator assigns.

Each user within the subscription has a workspace ID that's used to reference any data requested within that person's account.

Finding the Structure:

```
#Request Subscriptions
subscriptions = requests.get(url='https://cloud.wago.com/api/core/subscriptions?api-version=1.0',
                             headers={'Authorization': 'Bearer ' + Auth_Token, 'api-key': API_Key})
print(subscriptions.json())
```

Figure 11: Wago Cloud GET method for subscriptions in Python

Like the POST function in figure 6, the GET function does not require a payload, since we are requesting information. Figure 12 shows the JSON output of that function, highlighting ID, name, and description of each subscription.

Now we'd need to store the ID for the subscription that houses the workspaces we're after (Figure 13). Keep in the mind the workspaces don't have to be within the same subscription, but you will have to reference the proper workspace IDs.

```
[
  {
    "id": "40a752ce-0ce4-4fcd-b520-5520ad028f9e",
    "name": "C. N.",
    "description": ""
  },
  {
    "id": "9988861a-be2a-426f-acd7-cbfbd26622b4",
    "name": "WAGO",
    "description": "Created in user registration"
  }
]
```

Figure 12: JSON output for subscription GET

We now have a list of all the Workspace IDs for each account within this subscription as well as the names that tie them together. This is essential in reading existing alarm data from the account we'd like to copy alarm data from.

The first order of business is now to find the two accounts we are interested in working with. Iterating through via For Loop, I am requesting the workspace ID by the name of the account (figure 16).

```
#Find The ID of the Profile you'd like to copy and the ID of the party you'd like to copy to
for i in range(0, len(Workspaces.json())):
    if Workspaces.json()[i].get('name') == 'Adam Reeve':
        AdamsID = Workspaces.json()[i].get('id')
        #print(AdamsID)
    if Workspaces.json()[i].get('name') == 'Joe Abdelmalak':
        JoesID = Workspaces.json()[i].get('id')
        #print(JoesID)
```

Figure 16: Collecting Workspace ID based on names

Finding available alarms:

Once a workspace ID is collected for both the receiving and accepting accounts, we will now search for all available alarms. The alarms are found within the Alarm configurations URL and the pertinent workspace ID (Figure 17). The output is shown below (Figure 18) which shows one test alarm called "Joe's other test"...don't ask what happened to the first one.

```
#Find the Alarm config and save it of the party you want to copy

AlarmConfig = requests.get(url= 'https://cloud.wago.com/api/alarmapp/alarmconfigurations?WorkspaceId='
                             +AdamsID + '&api-version=2.0',
                             headers= {'Authorization': 'Bearer ' + Auth_Token, 'api-key': API_Key})
print(AlarmConfig.json())
```

Figure 17: Alarm configuration GET request using Adam's workspace ID

```
[
  {
    "id": "ffbe791c-60e4-479e-bef4-82482a7d9d50",
    "general": {
      "alarmLevel": "Critical",
      "name": "Joe's other test",
      "description": "",
      "timeInterval": 1
    },
    "rules": [
    ],
    "hysteresis": {
    },
    "devices": {
    },
    "message": {
    },
    "recipients": [
    ],
    "type": "ValueBased"
  }
]
```

Figure 18: Collapsed JSON output of the Alarm config GET of Adam's account

Finding Value based alarms:

In the world of the Wago cloud, there are several different types of alarms. The main alarm type we will go through is a Value Based Alarm, which ties a variable to a specific variable range, in turn triggering that alarm.

To access all alarms under the "ValueBased" type class we will be using the code below. The for loop is simply used to create a list of Value based alarms (figure 19).

```
ValueBasedAlarms = []
ConnectionBasedAlarms = []
PlcStatusBasedAlarms = []
TimeIntervalBasedAlarms = []
TelemetryDataTimeoutBasedAlarms = []
AlarmFlagBasedAlarms = []

#Set Variables to Copy all the alarm configuration information

for i in range(0, len(AlarmConfig.json())):

    if AlarmConfig.json()[i].get('type') == 'ValueBased':
        if i == 0:
            ValueBasedAlarms = [AlarmConfig.json()[i]]
        if i > 0:
            ValueBasedAlarms.append(AlarmConfig.json()[i])
```

Figure 19: Variable declaration of alarm types & adding alarms to a Python list

Note: This configuration for finding all alarm types and storing them into a list can extend for all different alarm types.

It's also important to note that within this Wago cloud platform, many alarms are tied to a specific device within the workspace. If you want to link one alarm from one account to another, then you'll need to link them at the device level. And in order to do that, you'll need to find device IDs for each account.

Are you sick of finding things yet? Yeah me too...but hey that's the beauty of an automated process; you'll never have to touch the code again after this and it'll just run without a hitch.

Disclaimer: That last sentence was a bold-faced lie. Debugging never ends. You'll never rest.

Finding all device ids:

As mentioned before, we'll need to request all of the device IDs for my account as well as Adam's. Once that is done, we will store both ideas in their respective arrays. This will then allow us to map alarms from device to device (figure 20).

```
AdamsDevices = requests.get(url= 'https://cloud.wago.com/api/deviceapp/devices?WorkspaceId='
                               +AdamsID + '&api-version=1.0',
                               headers= {'Authorization': 'Bearer ' + Auth_Token, 'api-key': API_Key})

#print(AdamsDevices.json())
AdamsDeviceIDs= []
for i in range(0, len(AdamsDevices.json())):
    if i == 0:
        AdamsDeviceIDs = [AdamsDevices.json()[i].get('id')]
    if i > 0:
        AdamsDeviceIDs.append(AdamsDevices.json()[i].get('id'))

#print(AdamsDeviceIDs)

JoesDevices = requests.get(url= 'https://cloud.wago.com/api/deviceapp/devices?WorkspaceId='
                               +JoesID + '&api-version=1.0',
                               headers= {'Authorization': 'Bearer ' + Auth_Token, 'api-key': API_Key})

JoesDeviceIDs= []
for i in range(0, len(JoesDevices.json())):
    if i == 0:
        JoesDeviceIDs = [JoesDevices.json()[i].get('id')]
    if i > 0:
        JoesDeviceIDs.append(JoesDevices.json()[i].get('id'))
```

Figure 20: Finding and storing device IDs for Adam's account and Joe's account

Changing Alarm JSON format for a POST:

So we've reached the final steps of this process and like most things, there are a few difficulties. It turns out that you can't just take the alarm data from one account and post it to another account. The reason for this...formatting.

The data required for a POST is different than the data coming out of the GET. For value based alarms in particular, you need to map the device, variable structure (Collection), and the variable itself (tagKey) to relevant data of the new account (figure 21).

```
#Change the rules JSON block
for i in range(0,len(ValueBasedAlarms)):

    del ValueBasedAlarms[i]['id']
    ValueBasedAlarms[i]['rules'][0]['tag']['deviceId'] = AlarmConfig.json()[i]['rules'][0]['tag']['deviceId']
    ValueBasedAlarms[i]['rules'][0]['tag']['collectionKey'] = AlarmConfig.json()[i]['rules'][0]['tag']['collectionKey']
    ValueBasedAlarms[i]['rules'][0]['tag']['tagKey'] = AlarmConfig.json()[i]['rules'][0]['tag']['tagKey']
    #print(ValueBasedAlarms[i])

if ValueBasedAlarms:

    ValueBasedAlarmsPost = requests.post(url='https://cloud.wago.com/api/alarmapp/alarmconfigurations/valueBased?Workspace='
                                         +JoesID+'&api-version=2.0',
                                         headers={'Content-Type': 'application/json; charset=utf-8',
                                                  'Accept': 'text/plain', 'Authorization': 'Bearer ' + Auth_Token,
                                                  'api-key': API_Key},
                                         json=ValueBasedAlarms[i])
    print(ValueBasedAlarmsPost)
```

Figure 21: Changing data format for new account

With this new formatting you should now be able to print alarms from one account to the next. Some things to note with this program:

The mapping of one device's alarms to another device's alarms is not an automated process. This is currently done by accessing the structure of AlarmConfig and changing those data points by hand. Of course, this can be done in an automated way (but I'm not clever enough).

The hope for this documentation is to show some functionality within the Wago Cloud API as well as showing a general sequence of events for the code structure when dealing with our API. There may be other developers out there that are more familiar with best practices in terms of security or even logic (In fact there definitely are).

As I post this out in the world I'd love to hear feedback for the code or even the API itself. Wago is constantly investigating new features for our software and product base, so feel free to let me know what you think!