Dashboard Code Documentation

The purpose of this document is to explain in detail the python code that supports the dashboards in general. Its most important role is the extraction of parameters from raw contract data, but it also performs other functions such as sending data to a google sheet, sending automatic emails etc.

This document shall focus particularly on the 'open_trades_dashboard.py' code.
As detailed in the aws documentation, there are three scripts running at the moment- 'open_trades_dashboard.py', 'closing_trades_dashboard.py', and 'gmx_open_trading.py'.

'closing_trades_dashboard.py' differs from 'open_trades_dashboard.py' only in specifics pertaining to the fact that it is involved in closing instead of opening trades.
For instance, the name of the dashboard it is pushing to would be different, and the variables that it reads from would differ. However, the structure and data extraction logic would remain the same.
Similarly, the data extraction part of 'gmx_open_trading.py', is based on the dashboard code.
Thus, we shall be looking at the 'open_trades_dashboard.py' code.

The code is currently present in the 'Trading/Deployed_Code' folder of the main repo.

The code begins by importing the required modules. Following are the modules imported-

```
from flask import Flask, request
from gevent.pywsgi import WSGIServer

import requests

import json
import time
import datetime
from decimal import Decimal

import smtplib
import ssl
from email.message import EmailMessage
from tabulate import tabulate

from binance.client import Client

import gspread
from oauth2client.service_account import ServiceAccountCredentials
```

Modules:

Flask: For creating a web server.

WSGIServer: To use a more stable WSGI server instead of the default development server, which is not recommended in a production deployment.

Requests: For making http requests

JSON: For json data manipulation

datetime: For handling timestamps

smptlib: For sending emails

ssl: For secure connections while sending the emails

tabulate: To tabulate the data before sending it in the mail

Client: For getting data from binance exchange and making trades on binance

gspread: For interacting with google sheets

ServiceAccountCredentials: to authenticate with Google Sheets using a service account's JSON key file

---

The next part involves google sheets authentication

```
# for writing to gsheets dashboard

scopes = [
    'https://www.googleapis.com/auth/spreadsheets',
    'https://www.googleapis.com/auth/drive'
]

# api keys for google
creds = ServiceAccountCredentials.from_json_keyfile_name(
    'dashboard_key.json', scopes)
gfile = gspread.authorize(creds)
workbook = gfile.open('Open_Trades_Dashboard')
sheet = workbook.worksheet('Sheet1')
```

For a detailed reference with respect to the steps required for writing to google sheets via python, refer to the video used https://www.youtube.com/watch?v=hyUw-koO2DA.

As explained in the video, the segment uses the created service account's JSON key file for google sheets authentication, and defines the name of the Sheet to write to.

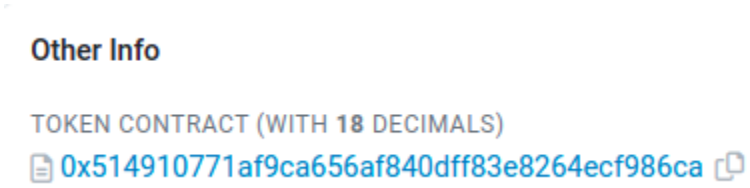
After defining the binance API keys and initializing a binance client object, the following code is used to calculate the value of the' exponent offset' variable.

```
# To obtain token's live 'decimals' values from ethplorer, to get the exponent_offset value later
link_exp_offset=int(requests.get("https://api.ethplorer.io/getTokenInfo/0x514910771AF9Ca656af840dff83E8264EcF986CA?apiKey=freekey").json()["decimals"])
uni_exp_offset=int(requests.get("https://api.ethplorer.io/getTokenInfo/0x1f9840a85d5af5bf1d1762f925bdaddc4201f984?apiKey=freekey").json()["decimals"])
wbtc_exp_offset=int(requests.get("https://api.ethplorer.io/getTokenInfo/0x2260fac5e5542a773aa44fbcfedf7c193bc2c599?apiKey=freekey").json()["decimals"])
weth_exp_offset=int(requests.get("https://api.ethplorer.io/getTokenInfo/0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2?apiKey=freekey").json()["decimals"])
```

The 'exponent offset' variable is used later in the code and is used for figuring the number (exponent of 10) by which the quantity variable needs to be divided.

It uses the ethplorer api to extract this value.
For instance, this is the 'link' token's page on etherscan https://etherscan.io/token/0x514910771af9ca656af840dff83e8264ecf986ca.

The value of the desired 'decimals' value can be seen here: 18.



The code thus extracts the value of the 'decimals' variable from the received json data.

Next, the list of trader addresses that are being followed are defined.

```
# will read from a file as no. of traders increase
profitable_traders_list = [
    '0xc97cfd2c3a3e61316e931b784bde21e61ce15b82','0xe2823659be02e0f48a4660e4da008b5e1abfdf29',

]
```

In this case they are hardcoded into a list (22 traders at the moment).
A better practice, especially as the number of traders increase, would be to read them from a file.
The list just makes the code simpler and more transparent to modify for now.

—

In this part, a flask application that handles incoming post requests is created.

```
app = Flask(__name__)

# specifies the route, and the http requests the the root will respond to

@app.route('/', methods=['POST'])
# function that will be executed when the route is called
def webhook():
```

'app=Flask(__name__)' creates an instance of the flask class, and assigns it to a variable called 'app'.
'@app.route('/'', methods=['POST'])' is a decorator that registers a function (the below webhook function in this case) as a handlet for a specific url and http method.

Here, the function 'webhook' will be called when a POST request is sent to the root URL of the app, used to receive data from other services/applications.

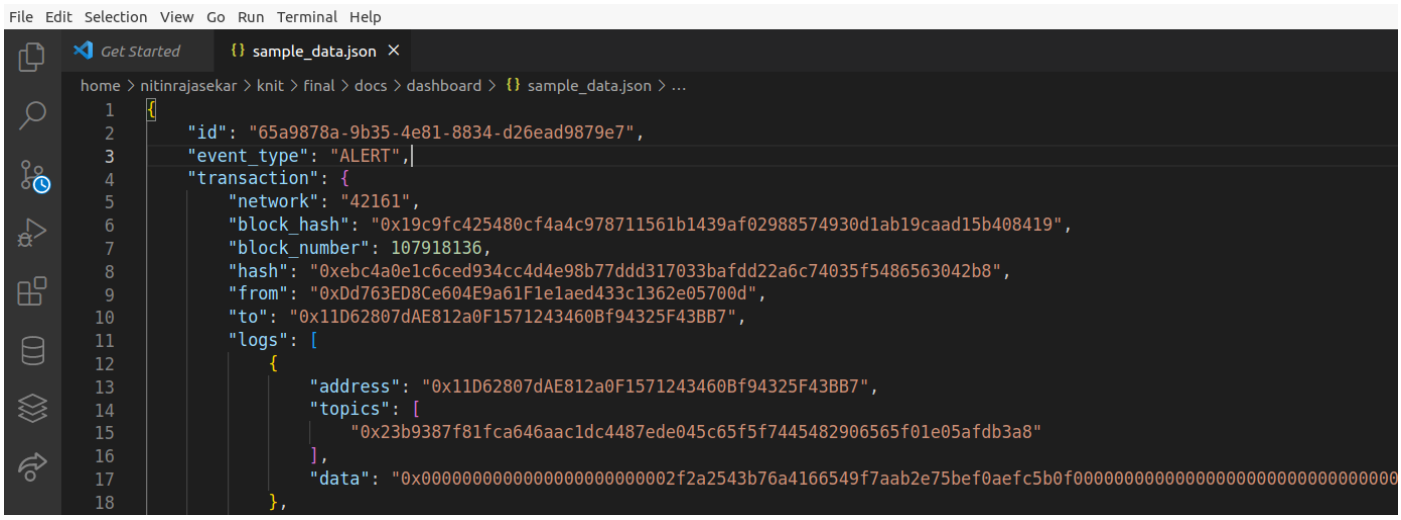
Next is the code to obtain the timestamp at which the request is received, ie. the time at which the trade was made.

```
time_var = time.time()
time_var = round(time_var)
dt_object = datetime.datetime.fromtimestamp(time_var)
# converting to IST from UTC
dt_object += datetime.timedelta(hours=5, minutes=30)
```

The variable time_var firstly gets the current time in terms of seconds from epoch, and rounds it to the nearest second.
dt_object then converts the 'seconds from epoch' value into a timestamp. As the inbuilt 'fromtimestamp' outputs in UTC by default, 5:30 is added to convert it to IST.

The code now begins with the actual extraction of information and values from the data received from the POST request.

The incoming POST request comes in with JSON data, which contains information about the transaction.
An example of the incoming data (to illustrate its structure and form) is in the sample_data.json file. It contains the data that was received when the transaction '0xebc4a0e1c6ced934cc4d4e98b77ddd317033bafdd22a6c74035f5486563042b8' was made.



```
data = request.get_json() # extracting json file
data = json.dumps(data)
# print(data) # can do this to see the raw json data that is being processed on below
data = json.loads(data)
```

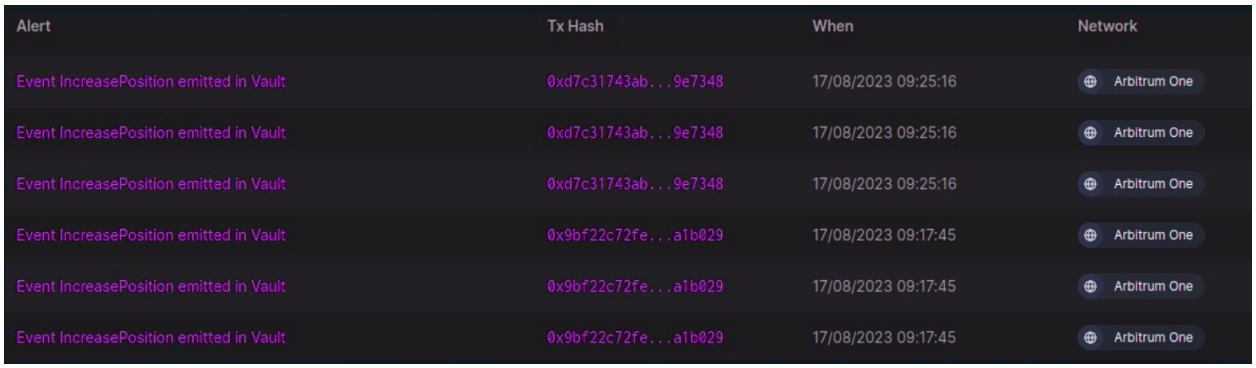
In the above segment of code, 'data=request.get_json()' retrieves the JSON data from the incoming POST request. The 'request' object is a part of Flask, and provides access to the incoming HTTP request data. The 'get_json()' method of the 'request' object parses the JSOn data and converts it into a python dictionary.

`tx_hash=data["transaction"]["hash"]` simply extracts the transaction ahsh attribute from the json data.

```
with open("tx_hash_store.txt", "r") as f:
    content = f.read()
    values = content.splitlines()
    if tx_hash in values:
        return 'OK', 200 # the transaction has already been processed before

    else:
        with open("tx_hash_store.txt", "a") as f2:
            f2.write(tx_hash+"\n")
```

The above section of the code is used to ensure that a given transaction (uniquely identified by its transaction hash) is processed once and once only.

Tenderly at times redundantly sends notifications for the same transaction more than once, as can be seen below



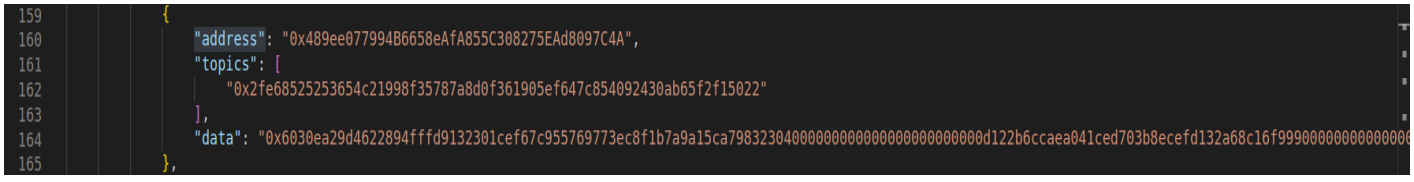
Alert	Tx Hash	When	Network
Event IncreasePosition emitted in Vault	0xd7c31743ab...9e7348	17/08/2023 09:25:16	Arbitrum One
Event IncreasePosition emitted in Vault	0xd7c31743ab...9e7348	17/08/2023 09:25:16	Arbitrum One
Event IncreasePosition emitted in Vault	0xd7c31743ab...9e7348	17/08/2023 09:25:16	Arbitrum One
Event IncreasePosition emitted in Vault	0x9bf22c72fe...a1b029	17/08/2023 09:17:45	Arbitrum One
Event IncreasePosition emitted in Vault	0x9bf22c72fe...a1b029	17/08/2023 09:17:45	Arbitrum One
Event IncreasePosition emitted in Vault	0x9bf22c72fe...a1b029	17/08/2023 09:17:45	Arbitrum One

So in order to resolve this, the snippet first checks if the received transaction hash is already in a file containing the list of previously processed hashes. If so, it exits the function.
If not, it writes the new hash in the file and continues, as the hash has not been already processed before.

Now the '`for i in data["transaction"]["logs"]:`' loop is used to loop through all the received events emitted.
In order to identify which of them is the 'IncreasePosition' event (which contains the parameters we are interested in) we take advantage of the fact that it is the only event with nine parameters.

Here, each parameter is represented by a 64 bit hexadecimal representation.

For instance, this is the represntation of IncreasePosition in the sample_data.json file.



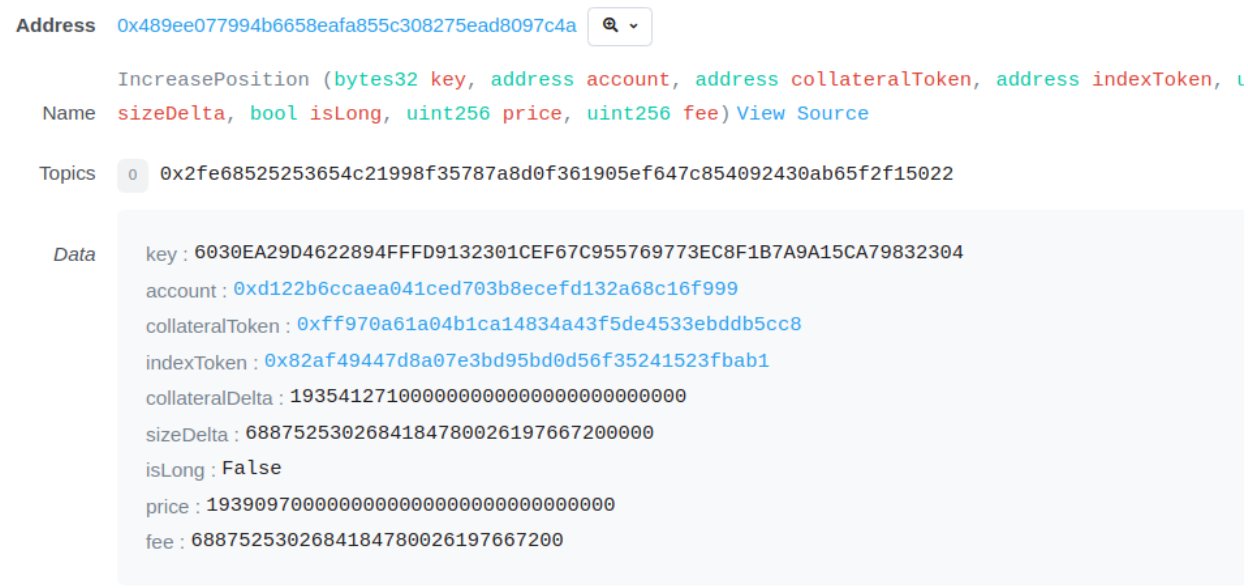
```
159         {
160             "address": "0x489ee077994B6658eAfA855C308275EAd8097C4A",
161             "topics": [
162                 "0x2fe68525253654c21998f35787a8d0f361905ef647c854092430ab65f2f15022"
163             ],
164             "data": "0x6030ea29d4622894fffd9132301cef67c955769773ec8f1b7a9a15ca798323040000000000000000000000000d122b6ccaea041ced703b8ecefd132a68c16f999000000000000000
165         },
```

The 'data' variable here is the one that consists of the 578 characters.

```
increase_position_found=1
string = i["data"]
string = string[2:]
# array of its parameter values
array = [string[i:i+64] for i in range(0, len(string), 64)]
```

The increase_position_found variable indicates that the IncreasePosition function has been succesfully found.
string holds the concatenated hexadecimal represntation of all the paramaters (the first 2 characters are removed as its 0x).

'array = [string[i:i+64] for i in range(0, len(string), 64)]' parses the string into an array of paramaters' hexadecimal representation, each of length 64.



The above screen screengrab is from the arbiscan page (logs), illustrating the parameters of the IncreasePosition event.

```
account = array[1]
account = "0x" + account[-40:]  # account
```

So here, the account param is got by getting the first param, and removing the first 40 chars (as it's 24 chars long, so the first 40 chars, all zeroes, of the 64 bit representation are removed).

In quite similar fashion, the col_token, token, isLong, col_delta, size_delta, and unit_price variables are extracted.
The variables that are in hexadecimal format (generally numeric values) are converted into decimal.
Leverage is calculated by dividing size_delta by col_delta.

```
if token == "0x82af49447d8a07e3bd95bd0d56f35241523fbab1":
    tok_str = "ETH"
    exponent_offset = weth_exp_offset
    max_precision=4

elif token == "0x2f2a2543b76a4166549f7aab2e75bef0aefc5b0f":
    tok_str = "BTC"
    exponent_offset = wbtc_exp_offset
    max_precision=5

elif token == "0xfa7f8980b0f1e64a2062791cc3b0871572f1f7f0":
    tok_str = "UNI"
    exponent_offset = uni_exp_offset
    max_precision=2

elif token == "0xf97f4df75117a78c1a5a0dbb814af92458539fb4":
    tok_str = "LINK"
    exponent_offset = link_exp_offset
    max_precision=2
```

Next, based on the token hash, we determine the currency parameters. The currencies supported on gmx v1 are eth, btc, link and uni.
The exponent offset has already been discussed earlier.

Now the 'max_precision' variable indicates the maximum number of decimal places the quantity can have, when expressed in a particular currency.
For instance, when we execute an order in say ETH, the quantity can have a maximum of 4 decimal places.
Attempting to buy say 1.23456 units of ETH (5 decimal places) would give an error - that the maximum precision has been exceeded.

```
if positionSide_var == "LONG":
    side_var = "BUY"
elif positionSide_var == "SHORT":
    side_var = "SELL"
```

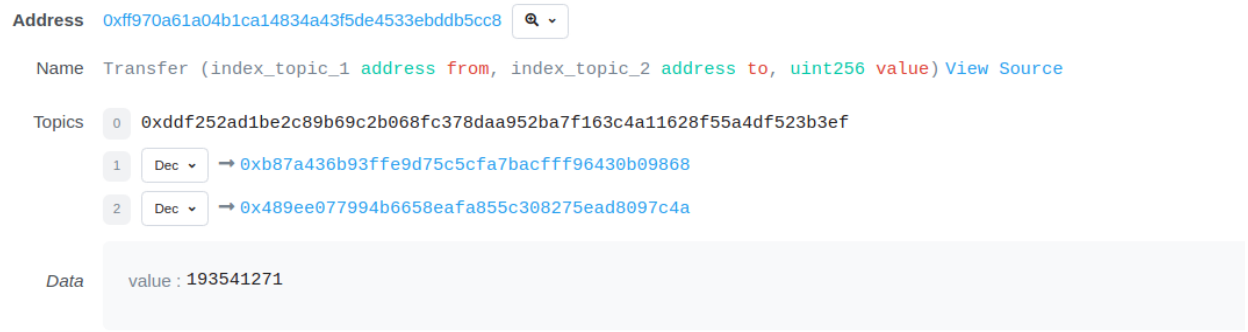
When increasing orders, indeed a long order and short order would correspond to a buy and sell respectively. This correspondence is switched in the decrease position code, where a long order decrease position would imply a sell order, and vice versa.

```
if positionSide_var == "LONG":

    for i in data["transaction"]["logs"]:
        if len(i["data"]) == 66 and len(i["topics"]) > 2 and i["topics"][2]
            string = i["data"]
            string = string[2:]
            # represents amount of index token
            new_quantity_var = float(int(string, 16))

    # get quantity of index token
    new_quantity_var = float(new_quantity_var)/(10**exponent_offset)
```

This part helps us calculate the quantity that the trader traded with during the transaction.

To do this, we make use of the 'Transfer' event that is emitted.

This is the event (from arbiscan again) for the sample transaction



The quantity here is 193.54 ETH. This quantity can thus be got from the value.

First, the Transfer event has to be identified, just like how we identified the IncreasePosition function.

To do this, we look for the function that has one param, whose second topic is '0x000000000000000000000000489ee077994b6658eafa855c308275ead8097c4a', and whose address param is the hash of the token being used.

```
for i in data["transaction"]["logs"]:
    if len(i["data"]) == 66 and len(i["topics"]) > 2 and i["topics"][2] == "0x000000000000000000000000489ee077994b6658eafa855c308275ead8097c4a" and
        string = i["data"]
        string = string[2:]
        # represents amount of index token
        new_quantity_var = float(int(string, 16))

# get quantity of index token
new_quantity_var = float(new_quantity_var)/(10**exponent_offset)
```

The Transfer event to consider will satisfy all of these conditions - we then get its value parameter.
Now to get the actual amount, we remove the first two chars (0x), and convert frmo hexadecimal to binary.

Now as we saw, the value displayed was `193541271,` although the quantity is 193.541271.
So to get the final quantity, we have to divide by a power of ten (determined by the already discussed exponent offset).
The above snippet for the case when the received position is a long one, and a very similar approach is utilized for short positions as well.

```
new_quantity_var = round(new_quantity_var, max_precision)

Is_in_list = "No"
if account in profitable_traders_list:
    Is_in_list = "Yes"
```

Next, the quantity is rounded up to the max_precision discussed earlier.
We also check whether the trader who executed the trade is part of our list of profitable traders.

```
# 12 attributes in each tuple on the dashboard
size = 12
send_list = [None] * size
send_list[1] = account
send_list[0] = data["transaction"]["hash"]
send_list[2] = positionSide_var
send_list[3] = side_var
send_list[4] = tok_str
send_list[5] = new_quantity_var
send_list[6] = unit_price
total_price = unit_price*new_quantity_var
total_price = round(total_price, 2)
send_list[7] = total_price
send_list[8] = str(dt_object)
send_list[9] = Is_in_list
```

The primary objective of the code is to get the extracted parameters onto the google sheets dashboard, and we begin doing this by first creating a list with all the required variables, whil we will later send to the gsheets dashboard.

Next, we calculate the liquidity. We intend to find the quantity of the currency available at the price the trader traded in (or better).
We do this by accessing the binance order book. We have already extracted the price of the currency when the trader made the trade.

```
if positionSide_var == "LONG":

    for ask in order_book["asks"]:
        if float(ask[0]) <= unit_price:  # only consider asks below or equal to the price
            # add the quantity to the total size
            liquidity += float(ask[1])
```

For instance here, we go through the order book, find the quantities available at asking prices less than or equal to that the trader purchased it, and take their sum.

We can then send the final list with all the extracted params to the gsheets dashboard

```
# send the data to the gsheets dashboard
sheet.append_row(send_list, table_range="A1:L1")
```

Next, if the trader's address is part of our list, we send an email notifying that a profitable trader has made a trade.

This was the video reference utilized for sending an email via python, containing complete instructions on setting up the required permissions on the 'from' account, and details regarding the code.
(https://www.youtube.com/watch?v=g_j6ILT-X0k)

```
if __name__ == '__main__':
    #app.run(host='0.0.0.0', port=443)
    http_server = WSGIServer(('', 443), app)
    http_server.serve_forever()
```

The final part of the code is responsible for starting the flask web application and making it listen for incoming requests.
To do this, the code uses the 'WSGIServer' from the 'gevent.pywsgi' module.
A WGSI server is a lightweight HTTP serverm and 443 in this case specifies the host and port to bind the server to.
Generally, the usual line to run a server would be something like 'app.run(host='0.0.0.0', port=443)'. However the default development server is not recommended in a production environment.
The final line, 'http_server.serve_forever()' starts the WSGIServer and makes it listen indefinitely for incoming requests.

—----------x—----------