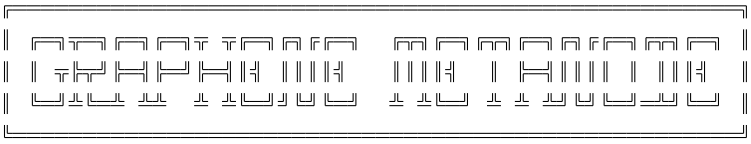
Graphene Metanode Tutorial

Version 6.11

March 26, 2022



Graphene Metanode is a locally hosted node for one account and several trading pairs that uses minimal RAM and CPU resources. It provides the necessary user stream and order book data for trading in a format one would expect from a centralized exchange API.

This project is phase one of a Hummingbot.io "connector" for Peerplays and other Graphene based Decentralized Exchanges.

Graphene Metanode presents an locally hosted API layer between the trader and Graphene; optimized to support multiple chains and currency pairs concurrently.

The Metanode was created because:

- A personal node requires expertise and resources.
- A long term connection to any node operator is improbable.
- Trusting a 3rd party blockchain API node can be risky.
- Some public API connections are faster than others.
- Querying a Graphene chain directly is not user friendly.

This tutorial was created to show you how to make a simple “pure market making” trading bot using the Metanode.

Before we can begin that, however, you must install the Graphene Metanode. To do this, please go to the next section, Installation

1 Installation

First, install and activate a virtual environment, replacing `<name of env>` with your chosen environment name:

```
pip3 install virtualenv
virtualenv <name of env>
. <name of env>/bin/activate
```

Next, install Metanode via pip:

```
pip3 install metanode
```

Finally, ensure Graphene Metanode was installed successfully by running the following command: `python3 -c 'import metanode'`

If that command finishes without errors, congratulations! Graphene Metanode is now installed!

Now that Graphene-Metanode is installed, we can begin creating our market making bot!

2 Creating our market making bot

The full code for this tutorial can be found here

To begin our market making bot, first open `graphene_constants.py` (which should be in your `site-packages/metanode` folder) and put your account name and preferred trading pairs in the correct `<chain>Constants` class. Then, create a file called `market_making.py` in the folder you created your environment in. After that, import the necessary modules and create global constants:

```
# STANDARD IMPORTS
import json
from multiprocessing import Process
from time import time, sleep

# METANODE IMPORTS
from metanode.graphene_auth import GrapheneAuth
from metanode.graphene_constants import GrapheneConstants
from metanode.graphene_metanode_client import GrapheneTrustlessClient
from metanode.graphene_metanode_server import GrapheneMetanode

PAIR = "BTS-HONEST"
CHAIN = "bitshares"
WIF = ""
AMOUNT = 0.1

CONSTANTS = GrapheneConstants(CHAIN)

# ...
```

- The first two imports, on line 2 through 4, import `json`, which will be used for handling our orders, `Process`, which will allow us to run the metanode in the background, and `time`, which we will use to time our orders.
- Then, on lines 6 through 9, we import various `metanode` modules that will allow us to sign transactions, access the user configuration, and use the data that the Metanode provides.
- The `PAIR` and `CHAIN` global constants can be changed, but be aware that `PAIR` *must* be in `GrapheneConstants.CHAIN.PAIRS` (`CHAIN` being your global variable).
- The `WIF` variable is currently a dummy, but in future chapters, will be the WIF for the account you entered in the `graphene_constants` file.
- `AMOUNT` is the amount of `BTS` (or the your configured core token) that will be placed in each order.

Now, we can begin creating the framework of our bot!

To start with, let's explain the notion of a "pure market making" bot:

- The bot gets the price of a given token.
- After getting that price, the bot calculates a 2% margin above and below the price, and places an order at each price.
- From there, if a certain amount of time has passed and the price has changed, those orders are cancelled, and the process begins again.
- If the orders are no longer there when we check, this means that somebody has filled our order. Then, because the person that filled the order had to place an order 2% away from the current price to fill our order, the bot (you) gained money.

By way of an analogy, if the going rate (current price) of 1 apple is $2, but I offer to buy apples for $1.96 (2% below) each and sell them for $2.04 (2% above) each, then any time someone gave me an apple for $1.96 and I sold it for $2.04, I would make a $0.08 profit. Of course, larger amounts mean larger profit.

Now that we understand the idea of pure market making, we can begin to implement the idea as follows:

```
# ...

def get_price():
    # Get price from metanode
    return 5 # This is a dummy price, and will become Metanode
'price' later.


def place_order(op, price):
    # Place order using GrapheneAuth.
    print(f"Placing {op} order at price {str(price)}")
    return "1.7.1234" # This is a dummy id, will become 'order_id'
later


def cancel_order(order_num):
    # Cancel order using GrapheneAuth.
    print("Cancelling order #" + str(order_num))
    return "1.7.1234" # Again, this is a dummy id.


def main():
    # Create an initial price and previous price of 0:
    price = pprice = 0
    # Initialize our order numbers as None:
    order1 = order2 = None
    # Main loop:
    while True:
        # Get the current price
```

```
        price = get_price()
        # Calculate 2% above and below:
        2percent = price * 0.02
        2above, 2below = price + 2percent, price - 2percent
        # Place our orders
        order1 = place_order("sell", 2above)
        order2 = place_order("buy", 2below)
        # Wait until 2 minutes have passed and the price has changed:
        start = time() # Current unix time
        while not ((time() - start < 120) and (price == pprice)):
            sleep(10) # Wait ten seconds
            pprice = price
            price = get_price() # Get the price
            # and try again
        # Cancel our now outdated orders:
        cancel_order(order1)
        cancel_order(order2)
        # and go through the loop again.

# ...
```

Most of the above code is fairly self-explanatory, due to the comments (marked by `#`), but here is a short breakdown:

- The first three definitions, `get_price`, `place_order`, and `cancel_order`, are dummies that will be replaced and fully documented in the next chapter.
- `main`, however, implements the pure market making strategy described above:
  - Get the current price.
  - Calculate 2% above and below the current price.
  - Place our above and below orders.
  - Wait until 2 minutes have passed and the price has changed:
    - Wait ten seconds.
    - Get the price.
    - And try again.
  - Cancel our now outdated orders
  - And go through the loop again.

Now that the framework is there, add these last two lines to the end of the file:

```
# ...
if __name__ == "__main__":
    main()
```

and run your code using

```
python3 market_making.py
```

If you get an error, check that your code matches this tutorial's exactly.

You should see

```
Placing sell order at price 5.1
Placing buy order at price 4.9
```

After that, the script hangs (Use Ctrl+c or Ctrl+\ to quit). This is because there are no live prices!

In the next section, however, we will fix that!

3 Adding the Metanode

Now that the framework of our market-making bot is complete, we can move on to making it usable! The first thing we must do to accomplish this is to start the Metanode when we start the script. To do this, create another definition at the top of all the ones made so far:

```
# ...
def start_metanode():
    metanode_instance = GrapheneMetanode(CONSTANTS)
    metanode_process = Process(target=metanode_instance.deploy)
    metanode_process.start()

    metanode = GrapheneTrustlessClient(CONSTANTS)
    blocktime = metanode.timing["blocktime"]
    latency = time.time() - blocktime
    while latency > 60:
        print("Waiting for Metanode... latency is " + str(latency))
        sleep(10)
        blocktime = metanode.timing["blocktime"]
        latency = time.time() - blocktime
# ...
```

- First, we create a Metanode instance, giving it `CONSTANTS` so that it knows the proper Graphene chain to use
- Then, we create a `Process` object that points to the `metanode_instance.deploy` method.
- Finally, we start the `metanode_process`. However, since `metanode_process` is a `Process` object, it does not block further code execution and we can carry on with our script with the metanode running in the background
- Now that the Metanode is started, we need to wait for it to spin up. To do this, we create a `GrapheneTrustlessClient` object and get the current `blocktime`, and see how long ago it was.
- Then we wait for `latency`, which is the difference between the blocktime and the current time, to be less than 60 seconds, meaning that the metanode has booted.

See the multiprocessing docs for more information on how Process works.

Since we now have the ability to start the Metanode, let's add this ability to `main`:

```
# ...

def main():
    start_metanode()
    # Create an initial price and previous price of 0:
```

```
# ...
```

Our Metanode is running! next, we need to actually use it. To start with, we are going to implement `get_price`:

```
# ...

def get_price():
    # Get price from metanode
    metanode = GrapheneTrustlessClient(CONSTANTS)
    price = metanode.pairs[PAIR]["last"]
    return price

# ...
```

- First, create a `GrapheneTrustlessClient` object, which allows us the use of the Metanode database.
- Then, all we have to do is access the `metanode.pairs` dictionary and index it by the current `PAIR`, which is defined at the top of the script.

Then `place_order`:

```
def place_order(op, price):
    # Place order using GrapheneAuth.
    print(f"Placing {op} order at price {str(price)}")
    auth = GrapheneAuth(CONSTANTS, WIF)
    order = json.loads(auth.prototype_order(PAIR))
    order["edicts"] = [
        {
            "op": op,
            "amount": AMOUNT,
            "price": price,
            "expiration": 0,
        },
    ]
    result = auth.broker(order)
    exchange_order_id = result["result"]["params"][1][0]["trx"][
        "operation_results"
    ][0][1]
    return exchange_order_id
```

This is slightly more complex than `get_price`, but it is still fairly easy to understand:

- To begin, we create a `GrapheneAuth` object, giving it `CONSTANTS` and `PAIR`, so that it knows what to trade with and on what blockchain.
- Then, using `json.loads()` — which loads a string to a python dictionary — we load a prototype order.
- However, merely a prototype order is not enough, so we add an `edict` to our order:
  - `op` is whether to `buy` or `sell`
  - `amount` is how much of the base token to buy or sell, this number is hard-coded in our case, so we just put our global variable.
  - `price` is how much of the `quote` token to buy/sell the `amount` of the base token for.
  - `expiration` would be the date at which the order would be automatically cancelled by the blockchain, but `0`, as we set it, means *no* expiration.
- Once we have an order, we can simply send it off to `auth.broker`, which will execute our order.
- After `auth.broker` executes our order, it returns a large dictionary of results, which we index to get our `exchange_order_id`.
- Finally, we return the `exchange_order_id`, which our market-making bot uses to keep track of its orders.

This step may seem overly complex, but `auth.broker` returns *everything* it gets from the blockchain, in case we wanted the `blocknum` or `blockid` that our order was placed at. (If you want to see all of the data, feel free to insert `print(result)` before the return statement.)

Finally, `cancel_order`:

```
def cancel_order(order_num):
    # Cancel order using GrapheneAuth.
    print("Cancelling order #" + str(order_num))
    auth = GrapheneAuth(CONSTANTS, WIF)
    order = json.loads(auth.prototype_order(PAIR))
    order["edicts"] = [
        {
            "op": "cancel",
            "ids": [order_num],
        },
    ]
    auth.broker(order)
    # No need to return the id
```

This, again, is fairly simple:

- The first two calls, defining `auth` and `order` are the same as in `place_order`.
- The `edict` we send off, however, is slightly different:

  - `op`, rather than being buy or sell, is now `cancel`.
  - Replacing `amount`, `price`, and `expiration` is now `ids`, which tells `auth.broker` which orders to cancel.

- Again, we send our order to `auth.broker`.