Trading alerts Documentation

Release 0.1.0

Kaishi Zhou

May 11, 2021

CONTENTS:

CHAPTER

ONE

TRADING ALERTS

This project intends to combine varying sources of OHLC financial data to provide a data-rich pipeline to automate various trading needs. One of the intended outcomes is to provide realtime alerts indicating best time to buy or sell a symbol (stock or crypto coin pair like bitcoin-usd).

As a minimum viable first version, this project intends to ingest cryptocurrency data with an Airflow pipeline and make it available via a basic (most likely Plotly) dashboard and Telegram alerts. The dashboard will aid exploring the coins and adding alerts to them. In this version, I intend to make RSI, TD-sequential and resistance/support line meteics available.

In a future version, I wish to add more indicators as I explore other trading strategies. This project is intended for self use but will be made available to small group of friends.

1.1 Why this project

Many such tools exist in the market which provide real time updates for a select set of traded signals. Majority of them are either too expensive for an invdividual user (above 100$ per month) or are limited to a limited set of free indicators. Both these factors were quite limiting for my use.

1.1.1 Learnings

1. Building dynamic graphs

Dynamic graphs are useful when your pipeline depends on varying number of inputs which cannot be fixed while designing the pipeline. For example - a sensor that monitors for a new key in a S3 bucket and hands of the processing to a new DAG once it finds that new file has been created. The use case for this project is discussed in more detail in the **`Internals`_** section.

Airflow provides a way to launch dynamic graphs via 3 operators - TriggerDagRunOperator, SubDagOperator, ExternalSensor operator. I chose to create dynamic graphs using the SubDagOperator because of the following reasons:

The SubDagOperator creates a hierarchial relationship between parent and child-dags and makes them available on UI via a drill down feature. So this not only allows you to glance at the parent dag status but also lets you “zoom-in” into the status of children dags if you want to. Not only this, but to use SubDagOperator you need a dag factory (a function or a file, which when executed returns a independent DAG) - this sort of brings it together for me.

TriggerDagRunOperator is ideal when you want to run a DAG independent of parent DAG once a condition is met and state history should be shared between the dags. These dags are not grouped under a single view in the UI.

The other alternative is ExternalSensor. It is a special operator whose function is to keep polling for a condition till its met or if it times out. These tend to occupy a worker till the conditoin turns true or operator times out.

So keeping all these things in mind, it was a better alternative to use SubDagOperator than the other two.

2. Historical and incremental DAGs

Inititally, I had thought of patiently polling the Binance API to collect the data. But I was skeptical of this approach because I needed minutely data for at least 200 crypto symbols and I was sure I was going to be banned within a day or two. (api limits to 1200 reqs per minute, with varying weightages of each API call)

Then I chanced upon historical archives of crypto data at 1m frequencies via https://data.binance.vision. I was actually hoping that maybe there would an unauthenticated API, which would let me collect data (safely). But things turned out better because after struggling through the HTML code I saw requests to S3 buckets. I was able to then take the bucket name from the "XHR" request and use the AWS cli commands that we used in pset-4, 5 to list the bucket contents. Turns out I could access the bucket without using AWS credentials! (yay.)

This is where I broke down the data collection architecture to use historical and incremental dags. Historical dags would collect data for new symbols for which no data wasn't collected earlier and incremental dags would come back every few minutes to top up the minute-wise data for coins that already had (using the Binance API). I was able to collect 1m interval data for about 5 coin pairs this way, which came around to 7Million rows of data.

3. Extending data upload in Postgres hooks

Airflow 2.0 does not have a data write functionality in its Postgres hooks yet. But, its possible to have this by grabbing the base class's connection URI and passing it to a pandas write statement. Having created as custom class around the PostgresHook and placing it in the "plugins" folder, let's you use it in any dag that may need it.

4. Postgres as a bookmark and a datastore

I use Postgres as the backend for Airflow pipeline. In this database, a *symbol* table keeps track of all symbols for which data extraction is going on. The *last_updated_at* columns is updated whenever a historical or incremental dag pulls data for the symbol. By keeping this column updated, the DAGs are able to calculate how many iterations of pulling day-wise data would be required to keep under the API throttle limits. Once this pipeline matures in usage, it would be easy to migrate it to Amazon's Redshift database (as it is build on Postgres) with least changes as compared to other databases like MySQL or MongoDB.

5. Idempotency

Airflow is an orchestration framework which does not provide options for idempotent operations for database or otherwise, out of the box. It is upto the end developer to do this. I have used delete SQL statements to remove data from the database before inserting into it. Architecturally, we only write to one table which holds data at 1m granularity, so the table would ideally need partitioning based on closing timestamp and symbol name. This way it would be possible to drop a partition and recreate it again - which as far as I've read is a better alternative than deleting select rows.

1.1.2 Features

- Automates data collection using Airflow. Internally, depends on Binance and Coingecko APIs
- Idempotent pipeline based on Airflow Postgres DB as backend
- Can scale by separating historical and incremental DAGs.
- Metadata about coin development pulled from Coingecko

CHAPTER

TWO

INSTALLATION

2.1 Development release

Clone repo and run `pipenv install`. This will install the necessary dependencies.

To setup metadata database: 1. Create Postgres metadata database

```
CREATE DATABASE airflow;
```

2. Create airflow user:

   ```
   CREATE USER USERNAME WITH PASSWORD '<password>';
   GRANT ALL PRIVILEGES ON DATABASE <database> TO USERNAME;
   ```

3. `export AIRFLOW\_HOME=$(pwd)`
4. Run: `airflow db init` to create the base config (which needs sqlite only)
5. Now change sql_alchemy_conn variable to a postgres db connection string, for which connection parameters were created above. And run: `airflow db reset`.

   This should create the necessary metadata database for data collection to proceed.

2.2 From sources

The sources for Trading alerts can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/kai490952010/trading_alerts
```

Once you have a copy of the source, you can install it with:

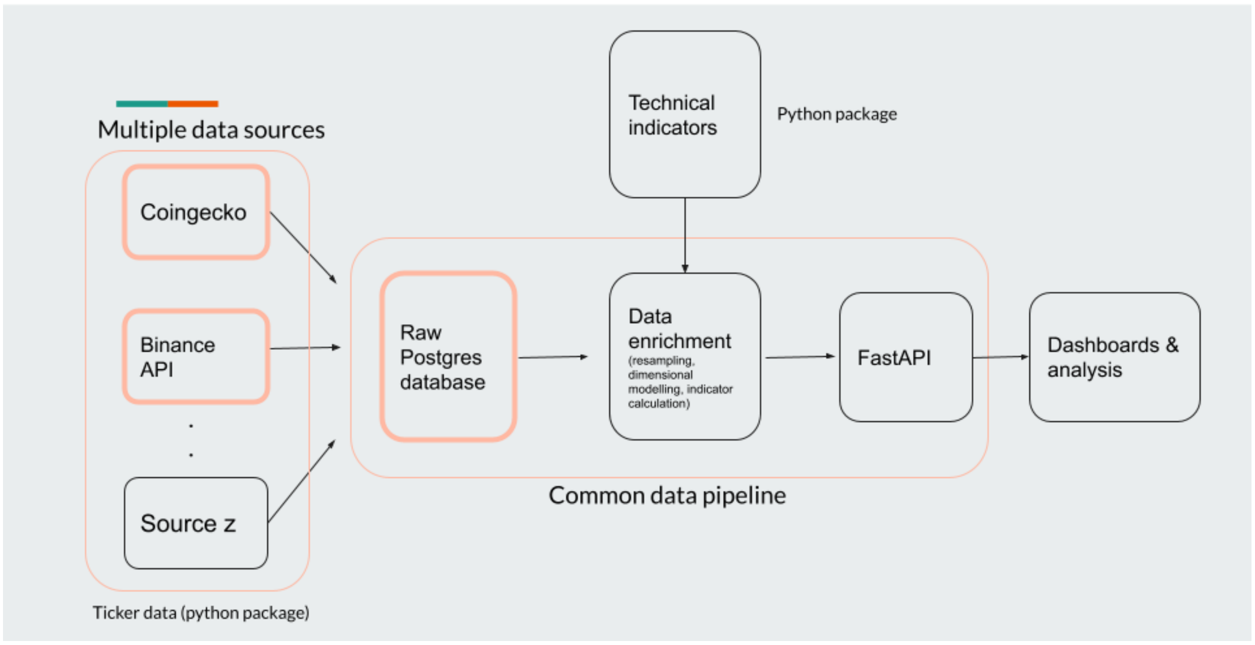
```
$ python setup.py install
```

CHAPTER

THREE

INTERNALS

3.1 Architecture



PS: Completed sections are marked in non-grey color. Rest are work in progress.

The project can be separated into 3 python packages:

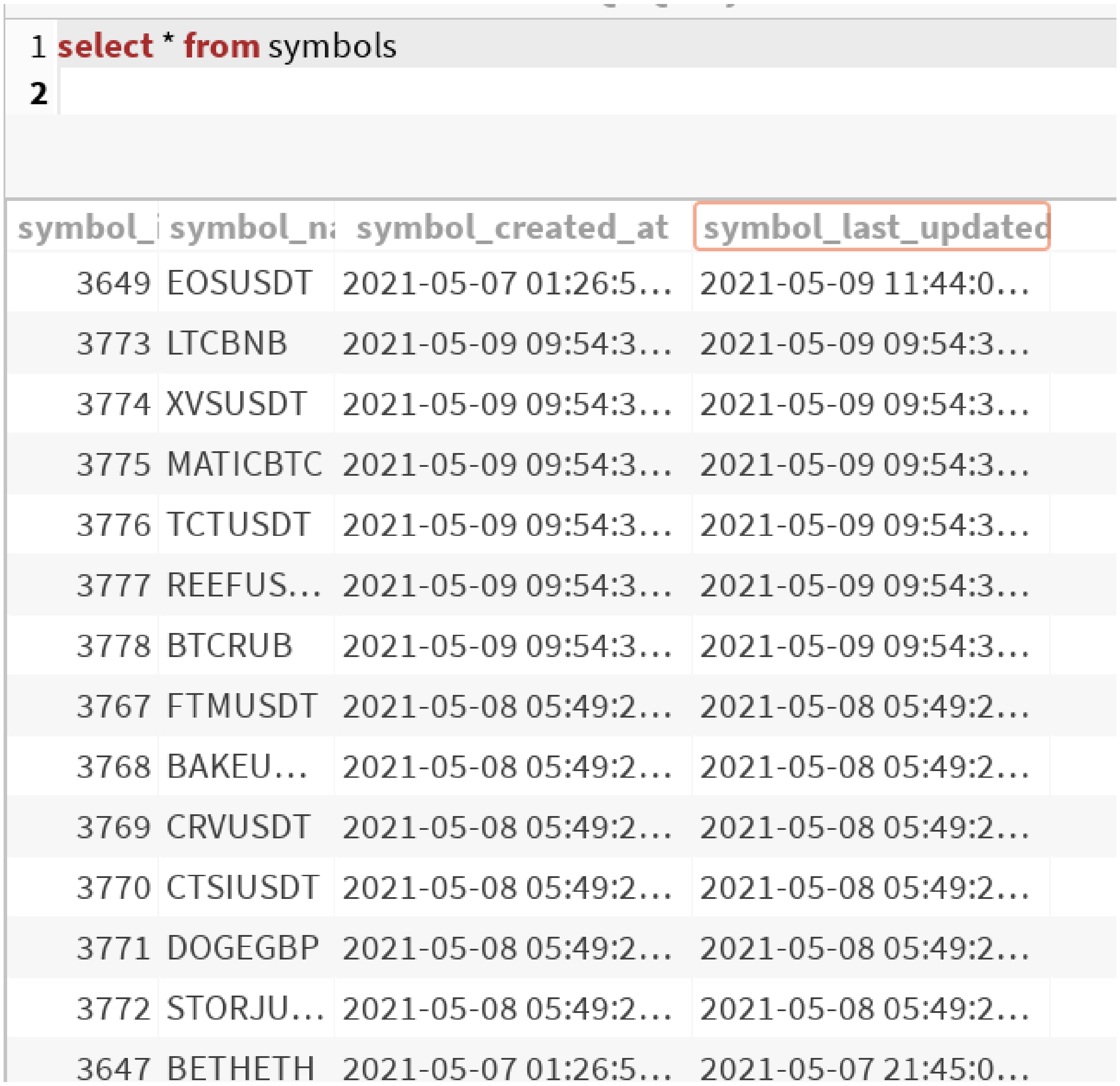
1. trading-alerts (current package) - this will contain the data pipeline package which will be fledged to collect and enrich the data. Include trading indicator calculation.

2. technical-indicators - all Python methods which calculate indicators including the ones outline in the introduction section will be housed here.

3. tickerdata - this will hold wrappers around APIs used for external data access. For example: with regards to crypto currency data - Binance & CoinGecko APIs will be housed within this package.

3.2 Execution model of data pipeline

The first dag that is run is the *get_symbols* DAG which pulls the top 100 cryptocurrency symbols by market cap from the coingecko API.



```
select * from symbols
```

symbol_i	symbol_na	symbol_created_at	symbol_last_updated
3649	EOSUSDT	2021-05-07 01:26:5...	2021-05-09 11:44:0...
3773	LTCBNB	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3774	XVSUSDT	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3775	MATICBTC	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3776	TCTUSDT	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3777	REEFUS...	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3778	BTCRUB	2021-05-09 09:54:3...	2021-05-09 09:54:3...
3767	FTMUSDT	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3768	BAKEU...	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3769	CRVUSDT	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3770	CTSIUSDT	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3771	DOGEGBP	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3772	STORJU...	2021-05-08 05:49:2...	2021-05-08 05:49:2...
3647	BETHETH	2021-05-07 01:26:5...	2021-05-07 21:45:0...

This table is updated daily (for now), and helps to capture a daily change in overall investment interest. If a crypto currency suddenly gains public interest it will improve by market cap and if it ranks within the top 100, I'd be interested.

The *last_updated_at* column and minutely data stored in *tickerdata_1m* table is used to calculate whether historical/incremental DAG should be run for the symbol.

The historical DAG runs a SQL command to find symbols which are listed in the *symbol* table but do not have any rows where *tickerdata_1m.close_time* is within the last month. It then hits the Binance S3 buckets to get monthly zip files of 1m ticker data for each symbol and use our custom Postgres hook to write to the table. The last thing that the DAG will do is update the *last_updated_at* column in the *symbol* table.

The incremental DAG will look at the *last_updated_at* column and calculate the number of Binance API requests that will be required to get data starting from *last_updated_at* timestamp to current timestamp. It then batches the jobs and creates a Pandas dataframe which will then be written to database. Again, the incremental DAG will update the *last_updated_at* column as well.

All timestamps in data processing are in UTC timezone.

3.3 Dynamic dags

Dynamic graphs are useful when your pipeline depends on varying number of inputs which cannot be fixed while designing the pipeline. For example - a sensor that monitors for a new key in a S3 bucket and hands of the processing to a new DAG once it finds that new file has been created.

Airflow provides a way to launch dynamic graphs via 3 operators - TriggerDagRunOperator, SubDagOperator, ExternalSensor operator. I chose to create dynamic graphs using the SubDagOperator because of the following reasons:

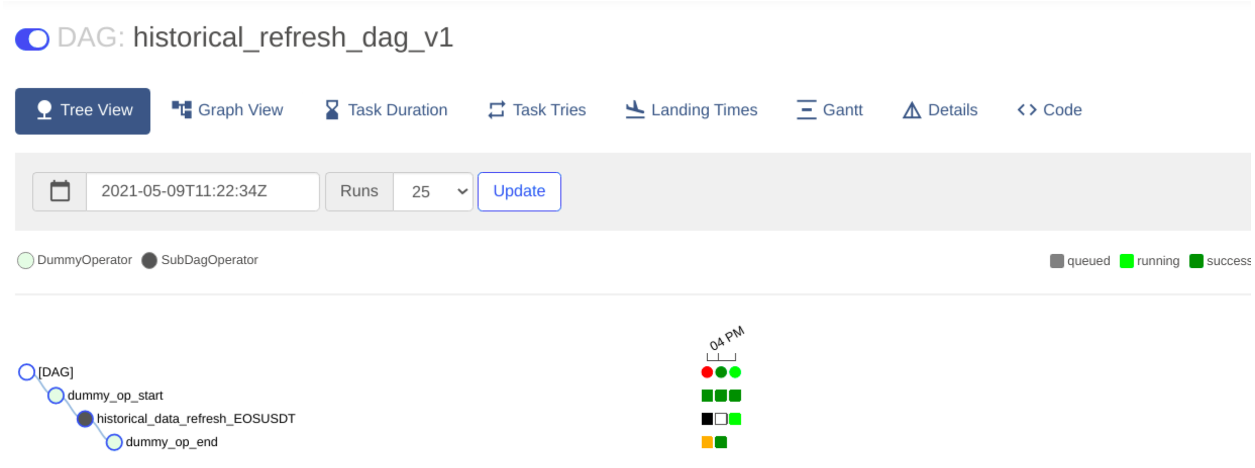
The SubDagOperator creates a hierarchial relationship between parent and child-dags and makes them available on UI via a drill down feature. So this not only allows you to glance at the parent dag status but also lets you "zoom-in" into the status of children dags if you want to. Not only this, but to use SubDagOperator you need a dag factory (a function or a file, which when executed returns a independent DAG) - this sort of brings it together for me.

TriggerDagRunOperator is ideal when you want to run a DAG independent of parent DAG once a condition is met and state history should be shared between the dags. These dags are not grouped under a single view in the UI.

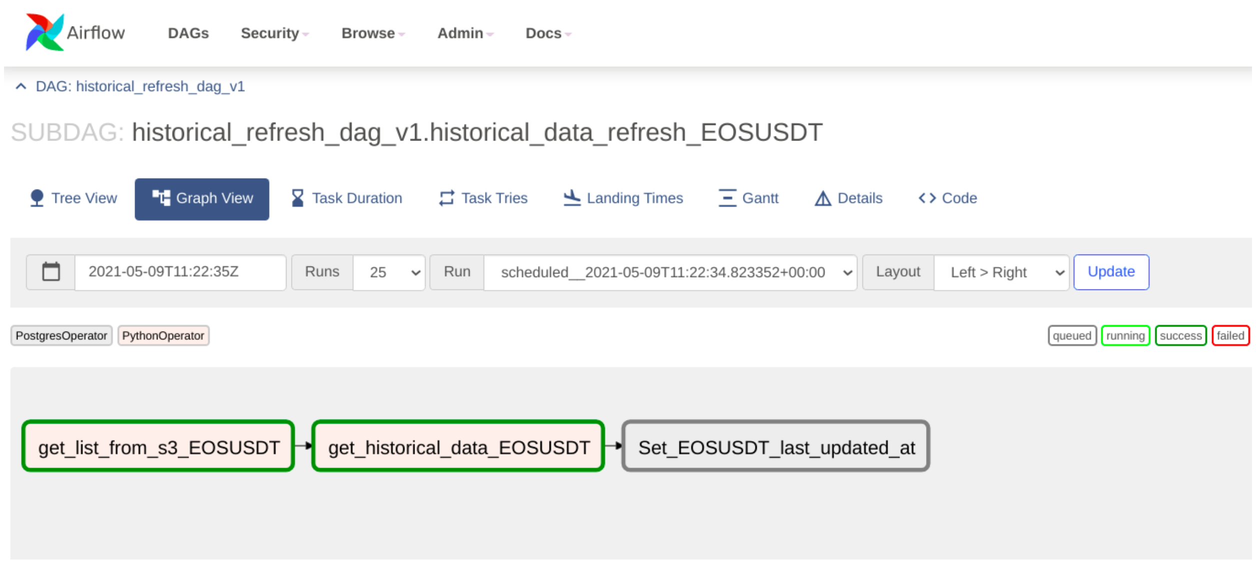
The other alternative is ExternalSensor. It is a special operator whose function is to keep polling for a condition till its met or if it times out. These tend to occupy a worker till the conditoin turns true or operator times out.

So keeping all these things in mind, it was a better alternative to use SubDagOperator than the other two.

Using SubDagOperator a parent DAG can create sub-dag to create historical data extraction DAGs for each coin. These are currently chained as I am operating in a laptop but could be run in parallel using a CeleryExecutor to reduce delays.



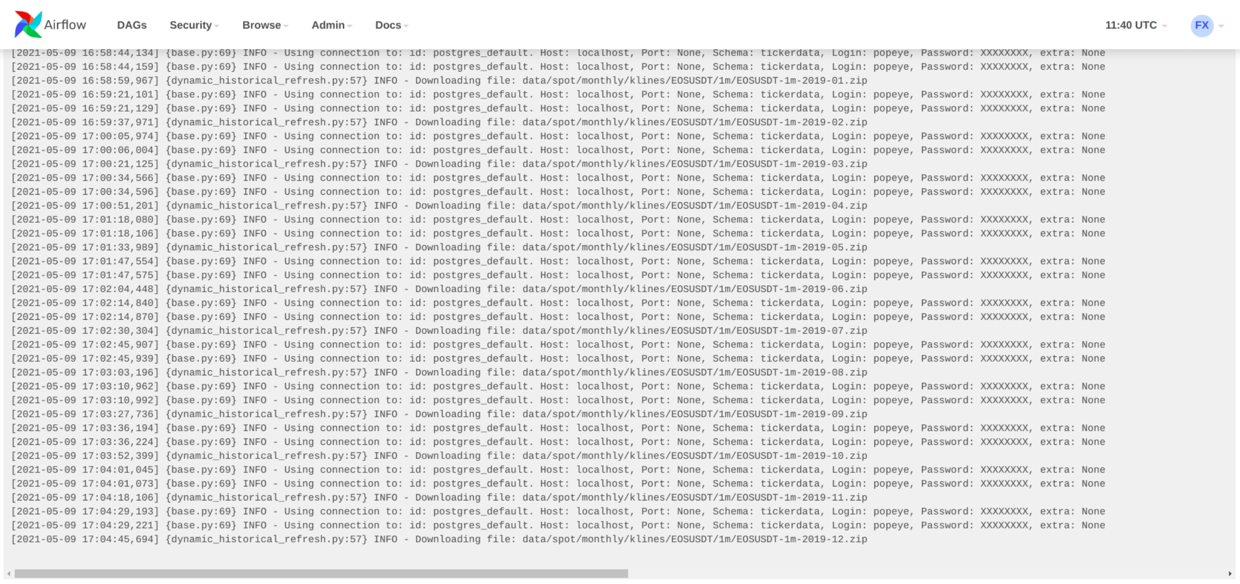
The above is a historical refresh for a coinpair taking place. Each sub DAG can be zoomed into the UI to see the progression of tasks within as shown below -



SUBDAG: historical_refresh_dag_v1.historical_data_refresh_EOSUSDT

2021-05-09T11:22:35Z Runs 25 Run scheduled__2021-05-09T11:22:34.823352+00:00 Layout Left > Right

Below are screenshots of the data pull logs from the Sub-Dag:



```
[2021-05-09 16:58:44,134] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 16:58:44,159] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 16:58:59,967] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-01.zip
[2021-05-09 16:59:21,101] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 16:59:21,129] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 16:59:37,971] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-02.zip
[2021-05-09 17:00:05,974] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:00:06,004] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:00:21,125] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-03.zip
[2021-05-09 17:00:34,566] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:00:34,596] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:00:51,201] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-04.zip
[2021-05-09 17:01:18,080] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:01:18,106] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:01:33,989] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-05.zip
[2021-05-09 17:01:47,554] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:01:47,575] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:02:04,448] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-06.zip
[2021-05-09 17:02:14,840] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:02:14,870] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:02:30,304] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-07.zip
[2021-05-09 17:02:45,907] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:02:45,939] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:03:03,196] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-08.zip
[2021-05-09 17:03:10,962] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:03:10,992] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:03:27,736] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-09.zip
[2021-05-09 17:03:36,194] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:03:36,224] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:03:52,399] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-10.zip
[2021-05-09 17:04:01,045] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:04:01,073] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:04:18,106] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-11.zip
[2021-05-09 17:04:29,193] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:04:29,221] {base.py:69} INFO - Using connection to: id: postgres_default. Host: localhost, Port: None, Schema: tickerdata, Login: popeye, Password: XXXXXXXX, extra: None
[2021-05-09 17:04:45,694] {dynamic_historical_refresh.py:57} INFO - Downloading file: data/spot/monthly/klines/EOSUSDT/1m/EOSUSDT-1m-2019-12.zip
```

3.4 Tickerdata package

Structurally, this package will contain a separate Python file for each data source it incorporates and all common functionality will exist modularised in a separate file.

In the *cryptocurrency* sub-module, two classes exist which provide different type of data - *BinanceAPI* and *CoingeckoAPI*. The former's structure is highlighted here.

```python
class BinanceAPI():
    def __init__(self):
        pass
```

(continues on next page)

(continued from previous page)

```
    def get_historical(self, symbol, start_ts=None, end_ts=None, granularity='1m'):
        pass
```

This class once instantiated will help get data between a start and end timestamp.

Similarly, a separate sub-module will be added for each of the NYSE and Shanghai stock exchanges.

3.5 Progress

Future versions of this package will include DAGs for dimensional modelling and indicator calculations. And, also a FastAPI rest interface to easily build dashboards from.

CHAPTER

FOUR

TRADING_ALERTS.DAGS PACKAGE

4.1 Submodules

4.2 trading_alerts.dags.dynamic_historical_refresh module

4.3 trading_alerts.dags.get_symbols module

4.4 trading_alerts.dags.incremental_data_refresh module

4.5 Module contents

CHAPTER

FIVE

TRADING_ALERTS.PLUGINS PACKAGE

5.1 Submodules

5.2 trading_alerts.plugins.custom_postgreshook module

5.3 Module contents

Top-level package for Trading alerts.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX