Tesla Inc.

Coding challenge

Instructions and guidelines

- You will have 1 day from the time of receipt of this challenge to respond back with a solution
- You are free to use any programming language as long as the code compiles
- Standard test modules (for testing your code) are typically written in Python/ R/ Javascript. If your solution is in a different language, please also include clear instructions on how to set up the compiler/ runtime
- If your solution uses non-standard packages, please include tests for all required dependencies, and install any additional/ missing packages needed before code execution begins
- You will need to use a SQL product (MySQL highly preferred) to test data upload/ download from your side
- If you are using a SQL product other than MySQL, please include clear instructions on what kind of database is used, and instructions on how to connect to it
- Clear commenting, as well as making your code modular is important
- Bonus points for good handling of edge and corner cases
- Please submit your solution as a compressed folder (ZIP) with all the files required to run the code
- Please include comments (and a write-up if possible) to explain your solution/ methodology. If the code fails to compile, your solution will be assessed purely based on the write-up/ comments
- Please refrain from sharing this coding challenge through private channels. The content of the attached documents/ files is the proprietary and confidential property of Tesla, Inc. and should be treated as such

Problem statement - Part I

The goals of this section are as follows:

- Fetch data from a remote source
- Transform JSON data into a relational form
- Uploading data in relational form into a SQL database

We will be using API endpoints to get the data we need to transform and load into our own database later on. For a quick introduction to APIs, please look here. For our challenge, we will be using the messari API. The API documentation for the messari API is available here.

API calls can be run through standard command line interfaces, or a program like Postman. You are encouraged to use a package/ program that best suits your needs/ solution, but ensure that your solution is platform independent.

Step 1: We will begin by getting data on the different markets that are covered by the messari API by running the following command

curl --get https://data.messari.io/api/v1/markets This returns the following JSON output


```
{
"status":{
        "elapsed":23,
        "timestamp":"2021-05-10T09:05:32.876302713Z"
        },

"data":[{
        "id":"0006addf-d3d2-491a-8a66-36236445c527",
        "exchange_id":"6efbfc8a-18c9-4c6f-aa78-feea100521cf",
        "base_asset_id":"3f548a16-866f-4e2c-b3e9-5df6d32d9083",
        "quote_asset_id":"51f8ea5e-f426-4f40-939a-db7e05495374",
        "class":"spot",
        "trade_start":"2020-08-30T16:40:06Z",
        "trade_end":null,
        "version":1,
        "excluded_from_price":false,
        "exchange_name":"Uniswap (v3)",
        "exchange_slug":"uniswap",
        "base_asset_symbol":"UOS",
        "quote_asset_symbol":"USDT",
        "pair":"UOS-USDT",
        "price_usd":null,
        "volume_last_24_hours":null,
        "deviation_from_vwap_percent":null,
        "last_trade_at":null
        }, ...
```

Task 1: The first task is to load the results of the API call into a variable. Write a function that executes the API call to **retrieve 40 markets** and load the results into a variable.

Task 2: The next task is to convert the data from JSON format into tabular/ relational form. Given a JSON object and a a list of specified fields, write a function that converts the JSON object into tabular form, selects only the specified fields and sets column headers from the list (in the same order used on the list).

List of fields:

```
[ 'id', 'exchange_name', 'base_asset_symbol', 'last_trade_at']
```

Hint: Consider using data frames

Task 3: For these markets, **ONLY** keep those that have had a trade volume_last_24_hours. If there is more than one market per base asset (base_asset_symbol), keep the one with the larger volume.

Task 4: Next, we need create an SQL table named ‘market_info’ in our database and upload the data above to the market_info table.

Bonus: Think about sensible data types for the different columns.

Given some database credentials, a table_name and some data, write a function that

- Creates a table in the database and names it according to the table_name (market_info in this case)
- Takes in the data (in relational/ tabular form) and uploads it onto the market_info table

Hint: Your code should check if table_name *exists in the database already. Because if the table exists, we don’t want to overwrite data on the table. Also, we do not want to add duplicates!*

Step 2: The next step would be to get the latest information on all traded assets:

https://data.messari.io/api/v2/assets

This API call produces the following output (output limited to suit explanation) :


```
{
"status": {
            "elapsed":475,
            "timestamp":"2021-05-10T09:29:03.089703863Z"},
"data": [{
          "id": "1e31218a-e44e-4285-820c-8282ee222035",
          "symbol": "BTC",
          "name": "Bitcoin",
          "slug": "bitcoin",
          "profile": {...},
          "metrics": {...}
},
...]
}
```


Please read the api docs for info on which assets are being returned and a list of all fields.

Task 1: Similar to Task 1 in Step 1, **fetch the information for 30 assets** via an API call and load the results of this API call into a variable.

Task 2: Convert the JSON output into the tabular form needed to upload into our database. For each asset, extract data according to a specified list of fields.

List of fields:

```
[ 'id','symbol','category','price_btc','price_usd']
```

Add the following extra fields:

```
Timestamp := the timestamp of the API call
Status := 0
```

An example is shown below:

id	symbol	category	price_btc	price_usd	timestamp	status
1e3121...	BTC	Payments	1	58047.8	2021-05-10 11:58:06.328816+02	0

Task 3: Only keep those assets, which are also a base asset of the 40 markets extracted in Step 1 and upload this data to a new table called asset_info.

The API call is a snapshot of current asset information and so the output obtained is dependent on the time at which the API is called.

Problem Statement - Part II

The goals of this section are as follows:

- Fetch data on regular intervals (from the same data source)
- Incrementally upload new data and flag anomalous behavior

As an extension of Part I, we will now :

- run the API call in Step 2 (current asset info) once every minute and append the data into the asset_info table. The API call should return new data at every call, as asset prices keep changing.
- as you get new data, decide whether the new data point is anomalous or not by setting the `Status` field equal to 1.

  Things to consider : You might want to think about what can be considered 'anomalous' data. Please DO NOT use any complex algorithms and keep the logic simple.

This section is being left intentionally open-ended because of the multitude of design choices available.

Bonus points if you consider database design principles like normalization, indexing, etc. as part of your implementation.

Task 1: Given a frequency (in minutes) of updating that defaults to 1 minute, encapsulate your code from Part I within a function such that it continuously executes at the specified frequency.