TDAmeritrade Streaming Data API

Streamer is a TD Ameritrade application that serves streaming market data internal and external clients of TD Ameritrade. In this tutorial, we will see how to create a connection to this application using python and have data streamed directly into our database. When we think about the steps necessary to get up and running, they can be boiled down to two significant actions:

1. Make a POST request to the User Principals endpoint, so that we can get the necessary info to log in.
2. Create a connection to the Streamer Application and subscribe to a data stream.

Additionally, I will be adding some extra steps that show us how to take the data we receive and insert it into a SQL database.

For those of you who would like to read some of the documentation, please feel free to visit the links provided. As a forewarning, all the examples shown in the literature will be coded using JavaScript.

- **Streamer Documentation:**
  - https://developer.tdameritrade.com/content/streaming-data (https://developer.tdameritrade.com/content/streaming-data)
- **User Principals Documentation:**
  - https://developer.tdameritrade.com/user-principal/apis/get/userprincipals-0 (https://developer.tdameritrade.com/user-principal/apis/get/userprincipals-0)

Libraries Needed:

The first section is dedicated to some of the libraries we will need to get started. In this tutorial, we will be working with JSON strings, so we will need to import the `JSON` library so we can effectively parse and encode our data that we either receieve or send in requests. We will also need the `urllib` library so we can URL encode some of our JSON requests that we will be passing along to the Streamer application.

Additionally, we will be working with datetimes for a little bit, and to parse those datetimes and convert them into the right values we will need to use the `dateutil` and `datetime` library. There will be some more libraries that I will be importing, but I will save those for when we arrive at a later part in the tutorial.

In [55]:
```python
import urllib
import json
import requests
import dateutil.parser
import datetime
from datetime import datetime
from TdAmeritradeStream import TDAuthentication
from config import password, account_number, client_id


# Alternative method for storing login info, if you choose not to use a config file.
# --------------------------------------
# password = 'YOUR PASSWORD'
# account_number = 'YOUR ACCOUNT NUMBER'
# client_id = 'YOUR CLIENT ID'
# --------------------------------------
```

Authentication

Before we can make our POST request to the User Principals endpoint, we will need to get an access token that we can send along in our request. Specific endpoints inside the TDAmeritrade API require that we authenticate ourselves before we use them. This endpoint contains sensitive information like account numbers so we will need to pass along an access token.

Now, I have dedicated a video on my channel where I explain how to authenticate ourselves using a combination of the `splinter` library and `urllib` library to do a few things.

1. Login to our account, and accept the terms and conditions.
2. Grab the access_code needed for the authentication endpoint.
3. Request the authentication endpoint to retrieve an access token.
4. Return that access token to the user.

If you would like to see how this works in more detail, please follow the link provided below to watch the video. If you're already familiar with the process, then you can skip ahead to the next section.

- Link to video:https://youtu.be/sVA0PeuDE4I (https://youtu.be/sVA0PeuDE4I)

Below I have provided the code necessary to grab an authentication token. **However, please note that this function does require the use of a ChromeDriver, so you will need to provide the path that points to the chrome driver.**

```
In [56]: import time
         from splinter import Browser

         def get_access_token():

             # define the location of the Chrome Driver - YOU MUST CHANGE THE PATH SO IT POINTS TO YOUR CHROMEDRIVER
             executable_path = {'executable_path': r'YOUR_OWN_CHROMEDRIVER_PATH'}

             # Create a new instance of the browser, make sure we can see it (Headless = False)
             browser = Browser('chrome', **executable_path, headless=True)

             # define the components to build a URL
             method = 'GET'
             url = 'https://auth.tdameritrade.com/auth?'
             client_code = client_id + '@AMER.OAUTHAP'
             payload = {'response_type':'code', 'redirect_uri':'http://localhost/test', 'client_id':client_code}

             # build the URL and store it in a new variable
             p = requests.Request(method, url, params=payload).prepare()
             myurl = p.url

             # go to the URL
             browser.visit(myurl)

             # define items to fillout form
             payload = {'username': account_number,
                        'password': password}

             # fill out each part of the form and click submit
             username = browser.find_by_id("username").first.fill(payload['username'])
             password = browser.find_by_id("password").first.fill(payload['password'])
             submit   = browser.find_by_id("accept").first.click()

             # click the Accept terms button
             browser.find_by_id("accept").first.click()

             # give it a second, then grab the url
             time.sleep(1)
             new_url = browser.url
             parse_url = urllib.parse.unquote(new_url.split('code=')[1])

             # close the browser
```

```
    browser.quit()

    print("Pulled Code, grabbing access token.")

    # THE AUTHENTICATION ENDPOINT

    # define the endpoint
    url = r"https://api.tdameritrade.com/v1/oauth2/token"

    # define the headers
    headers = {"Content-Type":"application/x-www-form-urlencoded"}

    # define the payload
    payload = {'grant_type': 'authorization_code',
               'access_type': 'offline',
               'code': parse_url,
               'client_id':client_id,
               'redirect_uri':'http://localhost/test'}

    # post the data to get the token
    authReply = requests.post(r'https://api.tdameritrade.com/v1/oauth2/token', headers = headers, data=payloa
d)

    # convert it to a dictionary
    decoded_content = authReply.json()

    # grab the access_token
    access_token = decoded_content['access_token']
    headers = {'Authorization': "Bearer {}".format(access_token)}

    return headers
```

There is another function we need to login correctly. This function takes a DateTime value and converts it into milliseconds, the Streamer Application does require one of the parameters to be in milliseconds, so this function will be leveraged to do just that.

In [54]:
```
def unix_time_millis(dt):

    # grab the starting point, so time '0'
    epoch = datetime.datetime.utcfromtimestamp(0)

    return (dt - epoch).total_seconds() * 1000.0
```

Grabbing Credentials from the UserPrincipals Endpoint

Let's start with the first step, which involves getting our credentials so that we can make our login request to the streamer application. First, define the endpoint, grab our headers using the `get_access_token` function we defined up above, define the parameters of our request and then finally make the request and call the `JSON` method to decode it.

As a side note, there are two parameters you need to pass through when defining the fields parameter. The `streamerSubscriptionsKeys` will contain the keys associated with your streamer account, and the `StreamerConnectionInfo` includes the information necessary to build our credentials request.

After decoding our response, we need to take the `tokenTimestamp` and convert it into milliseconds, so we will grab the value, turn it into a DateTime, and then pass it through our `unix_time_millis` function which will return the value in milliseconds.

Next, we will build our credentials dictionary, that we will pass along in our `login_request` . This dictionary follows what the documentation requires, and contains information like our accountID, appID, and our streamer token. If you'd like to learn more about this request, you can go to the following link:

- https://developer.tdameritrade.com/content/streaming-data#_Toc504640574 (https://developer.tdameritrade.com/content/streaming-data#_Toc504640574)

Once, we've built our credential dictionary, we can move to the next section.

In [ ]:
```python
# we need to go to the User Principals endpoint to get the info we need to make a streaming request
endpoint = "https://api.tdameritrade.com/v1/userprincipals"

# get our access token
headers = get_access_token()

# this endpoint, requires fields which are separated by ','
params = {'fields':'streamerSubscriptionKeys,streamerConnectionInfo'}

# make a request
content = requests.get(url = endpoint, params = params, headers = headers)
userPrincipalsResponse = content.json()

# we need to get the timestamp in order to make our next request, but it needs to be parsed
tokenTimeStamp = userPrincipalsResponse['streamerInfo']['tokenTimestamp']
date = dateutil.parser.parse(tokenTimeStamp, ignoretz = True)
tokenTimeStampAsMs = unix_time_millis(date)

# we need to define our credentials that we will need to make our stream
credentials = {"userid": userPrincipalsResponse['accounts'][0]['accountId'],
               "token": userPrincipalsResponse['streamerInfo']['token'],
               "company": userPrincipalsResponse['accounts'][0]['company'],
               "segment": userPrincipalsResponse['accounts'][0]['segment'],
               "cddomain": userPrincipalsResponse['accounts'][0]['accountCdDomainId'],
               "usergroup": userPrincipalsResponse['streamerInfo']['userGroup'],
               "accesslevel":userPrincipalsResponse['streamerInfo']['accessLevel'],
               "authorized": "Y",
               "timestamp": int(tokenTimeStampAsMs),
               "appid": userPrincipalsResponse['streamerInfo']['appId'],
               "acl": userPrincipalsResponse['streamerInfo']['acl'] }

userPrincipalsResponse
```

Let's define our login request and the data request. You'll notice that both of these requests have things in common. First, they both need to be in a dictionary, where the 'Master' key is `requests` which includes a list of all the requests that you will be making. Each request has a `service` value, which defines the service name we want. The second value is the `requestid` this will uniquely identify each request; the third value `command` sets the action you want to take on this endpoint.

This could be things like subscribing to the data or logging in. Additionally, we have an `account` and `source` key, which contains info that defines who we are. Finally, we have the `parameters` key, which will include a dictionary of all the parameters you would like to pass through in your request along with their associated value. Depending on the stream you're trying to subscribe to the values will change.

You'll notice, I've defined my login request and two data requests. The first data request retrieves the most actively traded stocks on the NASDAQ, and the second one retrieves future values for the associated symbol. Before you can send these request to the streamer, you will need to convert them to JSON strings using the `json` library.

```
In [ ]: # define a request
        login_request = {"requests": [{"service": "ADMIN",
                                      "requestid": "0",
                                      "command": "LOGIN",
                                      "account": userPrincipalsResponse['accounts'][0]['accountId'],
                                      "source": userPrincipalsResponse['streamerInfo']['appId'],
                                      "parameters": {"credential": urllib.parse.urlencode(credentials),
                                                     "token": userPrincipalsResponse['streamerInfo']['token'],
                                                     "version": "1.0"}}]}


        # define a request for different data sources
        data_request= {"requests": [{"service": "ACTIVES_NASDAQ",
                                     "requestid": "1",
                                     "command": "SUBS",
                                     "account": userPrincipalsResponse['accounts'][0]['accountId'],
                                     "source": userPrincipalsResponse['streamerInfo']['appId'],
                                     "parameters": {"keys": "NASDAQ-60",
                                                    "fields": "0,1"}},
                                    {"service": "LEVELONE_FUTURES",
                                     "requestid": "2",
                                     "command": "SUBS",
                                     "account": userPrincipalsResponse['accounts'][0]['accountId'],
                                     "source": userPrincipalsResponse['streamerInfo']['appId'],
                                     "parameters": {"keys": "/ES",
                                                    "fields": "0,1,2,3,4"}},
                                   {"service": "LEVELONE_FUTURES_OPTIONS",
                                     "requestid": "3",
                                     "command": "SUBS",
                                     "account": userPrincipalsResponse['accounts'][0]['accountId'],
                                     "source": userPrincipalsResponse['streamerInfo']['appId'],
                                     "parameters": {"keys": "/ES",
                                                    "fields": "0,1,2,3,4"}},
                                   {"service": "QUOTE",
                                     "requestid": "4",
                                     "command": "SUBS",
                                     "account": userPrincipalsResponse['accounts'][0]['accountId'],
                                     "source": userPrincipalsResponse['streamerInfo']['appId'],
                                     "parameters": {"keys": "SQ,AAPL,GOOG,MSFT",
                                                    "fields": str(','.join(fields_list))}}]}
```

```
# create it into a JSON string, as the API expects a JSON string.
login_encoded = json.dumps(login_request)
data_encoded = json.dumps(data_request)
```

This is probably the most challenging part of this tutorial, creating a WebSocket client that will allow us to connect to the streamer, send and receive messages, and connect and insert into the database. In this particular tutorial is to use the `pyodbc` library to connect and insert data into my database.

Before you begin this portion of the tutorial, you will need to install a few libraries.

- websockets
  - `pip install websockets`
- asyncio
  - `pip install asyncio`
- pyodbc
  - `pip install pyodbc`

The first thing we will do is create a new class object called `WebSocketClient` and define a new initialization method that will create a place holder for the connection object to our database, and a cursor object place holder for the cursor object associated with our database.

Next, I define a method to connect to our database; it will take a server value, a database name, and a driver so that we can connect to the SQL database. After a connection is made, it is attached to the WebSocket client object and additionally the cursor object as well. I also define a method that will take an insert query and a data tuple that will insert data into the database as we receive values from the Streamer Application.

That does it for the database portion, let's move on to the WebSocket. The WebSocket has a few components; the first will connect to the data stream, the second will be a mechanism to send our JSON strings (the message), and the other will be our receiver, which takes the response from the streamer, parses it and inserts it into the database.

Finally, we have a mechanism that will check the connection with the streamer every five seconds to verify we are still connected. If we are not connected a `ConnectionClosed` exception will be raised, and the program will halt.

A few additional notes, you may notice that a few of these functions leverage the `async` and `await` keyword. This might be your first time working with `coroutines` so I'll provide some documentation that will explain how these functions work.

- https://stackabuse.com/python-async-await-tutorial/ (https://stackabuse.com/python-async-await-tutorial/)
- https://docs.python.org/3/library/asyncio-task.html (https://docs.python.org/3/library/asyncio-task.html)

```
In [ ]: import websockets
        import asyncio
        import pyodbc

        class WebSocketClient():

            def __init__(self):
                self.cnxn = None
                self.crsr = None

            def database_connect(self):

                # define the server and the database, YOU WILL NEED TO CHANGE THIS TO YOUR OWN DATABASE AND SERVER
                server = 'SERVER_NAME'
                database = 'DATABASE_NAME'
                sql_driver = '{ODBC Driver 17 for SQL Server}'

                # define our connection, using windows authentiation.
                self.cnxn = pyodbc.connect(driver = sql_driver,
                                           server = server,
                                           database = database,
                                           trusted_connection ='yes')

                self.crsr = self.cnxn.cursor()

            def database_insert(self, query, data_tuple):

                # execute the query, commit the changes, and close the connection
                self.crsr.execute(query, data_tuple)
                self.cnxn.commit()
                self.cnxn.close()

                print('Data has been successfully inserted into the database.')

            async def connect(self):
                '''
                    Connecting to webSocket server
                    websockets.client.connect returns a WebSocketClientProtocol, which is used to send and receive me
        ssages
                '''

                # define the URI of the data stream, and connect to it.
        uri = "wss://" + userPrincipalsResponse['streamerInfo']['streamerSocketUrl'] + "/ws"
        self.connection = await websockets.client.connect(uri)

        # if all goes well, let the user know.
        if self.connection.open:
            print('Connection established. Client correctly connected')
            return self.connection


    async def sendMessage(self, message):
        '''
            Sending message to webSocket server
        '''
        await self.connection.send(message)


    async def receiveMessage(self, connection):
        '''
            Receiving all server messages and handle them
        '''
        while True:
            try:

                # grab and decode the message
                message = await connection.recv()
                message_decoded = json.loads(message)

                # prepare data for insertion, connect to database
                query = "INSERT INTO td_service_data (service, timestamp, command) VALUES (?,?,?);"
                self.database_connect()

                # check if the response contains a key called data if so then it contains the info we want to
insert.
                if 'data' in message_decoded.keys():

                    # grab the data
                    data = message_decoded['data'][0]
                    data_tuple = (data['service'], str(data['timestamp']), data['command'])

                    # insert the data
                    self.database_insert(query, data_tuple)

                print('-'*20)
                print('Received message from server: ' + str(message))

            except websockets.exceptions.ConnectionClosed:
                print('Connection with server closed')
                break


    async def heartbeat(self, connection):
        '''
            Sending heartbeat to server every 5 seconds
            Ping - pong messages to verify connection is alive
        '''
        while True:
            try:
                await connection.send('ping')
                await asyncio.sleep(5)
            except websockets.exceptions.ConnectionClosed:
                print('Connection with server closed')
                break
```

Now that we've built our client object, we can begin to make our request. You will need to install `nest_asyncio` to see the results in your notebook. After you've installed and imported the library, make sure to call the `apply()` method. Next, define a new instance of the `WebSocketClient` , and create a new `event_loop` so that we can pass along messages and receive requests.

Define a list of tasks, with the first one connecting to the client, the second logging on, and third sending the message. Notice that we also have defined some messages that will send us the received content. Start the loop and watch the data flow.

In [ ]:
```
import nest_asyncio
nest_asyncio.apply()


if __name__ == '__main__':

    # Creating client object
    client = WebSocketClient()

    loop = asyncio.get_event_loop()

    # Start connection and get client connection protocol
    connection = loop.run_until_complete(client.connect())

    # Start listener and heartbeat
    tasks = [asyncio.ensure_future(client.receiveMessage(connection)),
             asyncio.ensure_future(client.sendMessage(login_encoded)),
             asyncio.ensure_future(client.receiveMessage(connection)),
             asyncio.ensure_future(client.sendMessage(data_encoded)),
             asyncio.ensure_future(client.receiveMessage(connection))]

    loop.run_until_complete(asyncio.wait(tasks))
```