SOEN 6441 – Fall 2022

ADVANCED PROGRAMMING PRACTICES

PROJECT: STOCK ANALYSIS AND VISUALIZATION

SADATH ROSHAN – 40224159

MANPREET SINGH RANA - 40227463

CONTENTS

INTRODUCTION

The Stock Market plays a vital role in the financial growth of a nation. It provides a platform for buyers and sellers to share/trade equities(stocks) of publicly traded corporations. The stock market is extremely impacted by politics, news articles, nation's job market, import, export, and the company's plan for future developments, making it extremely volatile and complex. However, millions of dollars are invested and traded every working day of the week despite the risk of losses. The stock market has become a platform where all major business transactions take place, thus, studying and observing the status of the stock market is a particularly important aspect of investing.

In this project we focus on developing a Graphical User Interface based application which retrieves data of each stock in the stock market from a leading Application Programming Interface (API) provider which contains accurate and reliable data in financial market analysis to build functionalities supporting some of the major stocks of leading companies in the industry to visualize their stock values and their changes over time. We also design functionality to derive related information from the data obtained from the API. We also created a database to store and retrieve information on various stocks in the market for faster access to the information.

The design patterns, object relational structural patterns and the data source architectural patterns are studied to further gain a deeper understanding which aids in designing a specific method of implementation. We further developed test strategies for the verification and validation of the application, by manually testing individual functionalities on function specific test cases. This ensures seamless development of each layer in the application.

CHOICE OF LANGUAGES AND API DATA FORMAT

After careful consideration of the project's design requirements, we concluded that Python Programming language would provide easier implementation for the application.

Reasons for choosing Python as the main development language:

- Python is a very high-level language with a shallow learning curve which enables us to learn various requirements of the implementation faster.
- It is interpreted which allows us to dynamically develop and test code.
- It has a massive number of libraries with extensive documentation for each.
- It supports database design and development.
- It has libraries that support basic GUI design like Tkinter, wxPython and JPython, which is extremely crucial to the project.

For designing the main database of the project, we use MySQL as this provides support to connect to the server and execute parameterized queries using python due to the presence of a package in python 'mysql-connector-python'

Alpha Vantage – One of the most popular API providers among investors and traders, tracks real-time stock market data and uploads it to their API from where it is publicly accessible. They have a huge database with over one billion datapoints, with sub datasets for commodities, forex, indexes, equities, cryptocurrency along with their individual information. The API Timeseries data, along with latest news, company profiles, Economic indicators such as GDP, CPI, Inflation, Unemployment rate and many more.

Access to the API is free for every user, however, there are limitations to the number of API calls that can be made which is a maximum of 5 requests per minute and 500 requests in total per day. This restricts the extent to which data can be extracted from the API. Alpha Vantage also provides Technical Indicators to help predict stock values, but most are premium features and out of our project's scope.

API: [Alpha Vantage Query for GOOGL intraday data in 15-minute intervals]


```
{
    "Meta Data": {
        "1. Information": "Intraday (15min) open, high, low, close prices and volume",
        "2. Symbol": "GOOGL",
        "3. Last Refreshed": "2022-11-11 20:00:00",
        "4. Interval": "15min",
        "5. Output Size": "Full size",
        "6. Time Zone": "US/Eastern"
    },
    "Time Series (15min)": {
        "2022-11-11 20:00:00": {
            "1. open": "96.1500",
            "2. high": "96.2000",
            "3. low": "96.0800",
            "4. close": "96.2000",
            "5. volume": "4551"
        },
        "2022-11-11 19:45:00": {
            "1. open": "96.1599",
            "2. high": "96.1599",
            "3. low": "96.1599",
            "4. close": "96.1599",
            "5. volume": "413"
        },
        …
        …
        …
        …,
        "2022-09-30 04:15:00": {
            "1. open": "98.7900",
            "2. high": "98.7900",
            "3. low": "98.2400",
            "4. close": "98.2400",
            "5. volume": "6356"
        }
    }
}
```

SOFTWARE DEPENDENCIES AND TOOLS

Programming Language Requirements:

- Python: 3.10.2
- MySQL: 8.0.31

Python Libraries Required:

- Backend Development / Business Logic
    a. mysql.connector : pip install mysql-connector-python
    b. requests : pip install requests
    c. os
- Front End Development / Graphical User Interface
    a. tkinter : pip install tk
    b. ImageTK module of PIL library : pip install Pillow
    c. partial module of functools library : pip install functools
    d. plotly.graph_objects : pip install plotly
    e. kaliedo: pip install kaliedo
- Testing
    a. pytest : pip install pytest

Integrated Development Environment:

- Jupyter Notebook
- Visual Studio Code
- Python IDLE
- Spyder

DESIGN, OBJECT RELATIONAL STRUCTURAL AND DATA SOURCE ARCHITECTURAL PATTERNS

The core modules of our application include processing and storing intraday data at granularity of 15 mins for 4 stocks (Google, Amazon, Apple, and Meta). The intraday data in turn is used to derive hourly data, hourly data serves as input to daily data which in turn is used for obtaining weekly data. Using all this data visualizations are done. Since the application has a sequential flow and we do not want customers to interact and generate connections at will to the database, we have deployed a **Singleton Design Pattern,** such that only one database connection instance is generated per application run and shared across it.

We have used **Table Data Gateway (**Database.py) which acts as an interface and moves data b/w MySQL database and our application/business layer (System.py). The business layer fetches intraday data from the API in every run and stores it in a list data structure and sends it to TDG. The TDG unpacks this data, transforms, maps it into the required format, and writes it into the database. Besides, the TDG functionality consists of retrieving intraday data for calculating hourly data, and daily and weekly data.

TESTING STATERGY, TEST CASES AND OUTPUT

Since our application is a small subset of a production-scale fully operational stock visualization app, we have deployed manual unit and integration testing to test our application and validate three key points.

a. **Test case:** Checking if the application connects and receives data from the API.

Testing Strategy: This is confirmed by capturing the response from the API server. If the response code is 200, it ensures that the application's API-related functioning is correct.

b. **Test case:** Validating if the data from API is indeed being read by the application and through the TDG, is stored in the database.

Testing Strategy: Since every run of the application removes stale intraday data from the table and does a fresh reload, therefore returning a positive count of records from a particular stock table after completion of the intraday table load function confirms that data loading is also working fine.

c. **Test Case:** Data validation for intraday, hourly, daily, and weekly data.

Testing Strategy: Manually monitoring the JSON output from the API for specific values and comparing if these are stored correctly in the database for intraday data.

For hourly, daily, and weekly data calculated the expected outputs and asserted them with the actual outputs.

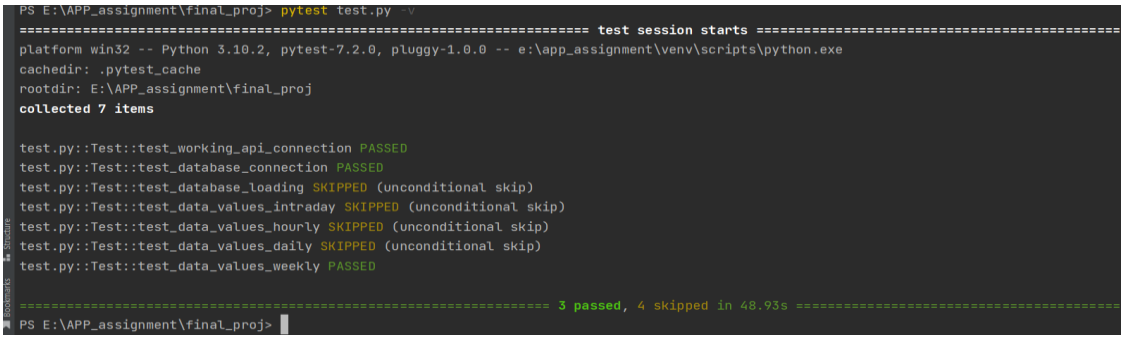
We have used **pytest testing framework** to test and validate our code

The test.py file to verify and validate the above given test cases is given below.

```
import Database as db
import System
import pytest
import requests


# The API allows 5 calls/min. The test cases annotated with @pytest.mark.skip
should be run only one at a time. Once
# a test case marked with @pytest.mark.skip is commented and run, please wait for a
minute to run the next test case
# to allow API call limit to be reset.

class Test:
```

```
    def test_working_api_connection(self):
        url =
'https://www.alphavantage.co/query?function=OVERVIEW&symbol=GOOGL&apikey=7NVAEAZYAX
PYKMZB'
        stock_data = requests.get(url)
        assert stock_data.status_code == 200

    def test_database_connection(self):
        db_obj = db.Database('localhost', 'root', "#abcd", 'soen6441', ['GOOGL'])
        out = str(type(db_obj.ser_connection))
        assert out == '<class
\'mysql.connector.connection_cext.CMySQLConnection\'>'

    @pytest.fixture()
    def database_object(self):
        db_obj = db.Database('localhost', 'root', "#abcd 'soen6441', ['GOOGL'])
        return db_obj

    @pytest.fixture()
    def system_object(self):
        sys_obj = System.System()
        return sys_obj

    @pytest.fixture()
    def intraday_dict(self, system_object):
        intraday_dict = system_object.get_intraday_data(['GOOGL'])
        return intraday_dict

    @pytest.mark.skip
    def test_database_loading(self, database_object, system_object, intraday_dict):
        connection = database_object.ser_connection
        count = database_object.read_query(connection, "select count(1) from
GOOGL")
        assert (count[0][0] >= 1000)

    @pytest.mark.skip
    def test_data_values_intraday(self, database_object, system_object):
        system_object.get_intraday_data(['GOOGL'])
        out_intraday_close = system_object.intraday_data['GOOGL']['2022-11-11
18:15:00']['Close']
        out_intraday_low = system_object.intraday_data['GOOGL']['2022-11-11
18:15:00']['Low']
        assert out_intraday_close == 96.24
        assert out_intraday_low == 96.2

    @pytest.mark.skip
    def test_data_values_hourly(self, database_object, system_object):
        connection = database_object.ser_connection
        system_object.get_hourly_data(database_object.ser_connection)
        out_hourly_open = system_object.hourly_data['GOOGL']['2022-11-11
18:00:00']['Open']
        out_hourly_low = system_object.hourly_data['GOOGL']['2022-11-11
18:00:00']['Low']
        assert out_hourly_open == 96.2
        assert out_hourly_low == 96.1
```

```python
@pytest.mark.skip
def test_data_values_daily(self, database_object, system_object):
    connection = database_object.ser_connection
    system_object.get_daily_data(database_object.ser_connection)
    out_daily_high = system_object.daily_data['GOOGL']['2022-11-11']['High']
    out_daily_low = system_object.daily_data['GOOGL']['2022-11-11']['Low']
    assert out_daily_high == 96.93
    assert out_daily_low == 93.706

@pytest.mark.skip
def test_data_values_weekly(self, database_object, system_object):
    connection = database_object.ser_connection
    system_object.get_daily_data(connection)
    out_weekly_high = system_object.weekly_data['GOOGL']['2022-11-04']['High']
    out_weekly_open = system_object.weekly_data['GOOGL']['2022-11-04']['Open']
    assert out_weekly_high == 97.03
    assert out_weekly_open == 96.33
```



SOFTWARE ARCHITECTURE DOCUMENT

1. INTRODUCTION

Studying the stock market involves understanding the trend in the market at any given point in time. One of the best ways to achieve this is by visualizing the market of any product in real time. This application enables data collection from the point of service (Alpha Vantage) along with local storage of the data in a database local to the user, thereby allowing faster read and write operations and extracts information at regular intervals of time from the obtained data.

1.2. OBJECTIVE

Our system aims to provide a tool for analyzing stock products of some of the world-famous corporations and provide their real time values against time as visualizations. With our application in place, investors and traders can observe the market trend and choose whether to invest in the product at the time.

1.3. SCOPE

The team is working intently to build a completely operational and fully viable platform for the market. The initial release of the product is complete with basic and crucial functionality. However, the evolution of the product will be subject to market demand and will be planned, developed and released incrementally with proper stakeholder involvement.

This Software Architecture Document applies to the entire Stock Analysis and Visualization application so far developed, however more functionalities for the application will be included in the next release as elaborated in the Future Plan section.

1.4. GLOSSARY AND ACRONYM LIST

Glossary:

- User: Any person using the software application to study the various stocks.
- Python: Interpreted, Object oriented programming language.
- MySQL: Local database server to store and retrieve data.
- TKinter: Python's library to build and deploy graphical user interfaces.
- Alpha Vantage: API provider for stocks, forex and cryptocurrency data.

Acronyms:

- API: Application Programming Interface.
- SQL: Structured Query Language
- GUI: Graphical User Interface
- CRUD: Create, Read, Update and Delete
- TDG: Table Data Gateway

2. NON-FUNCTIONAL REQUIREMENTS

High Performance: The system should be able to retrieve data, store, process and plot visualization in real-time

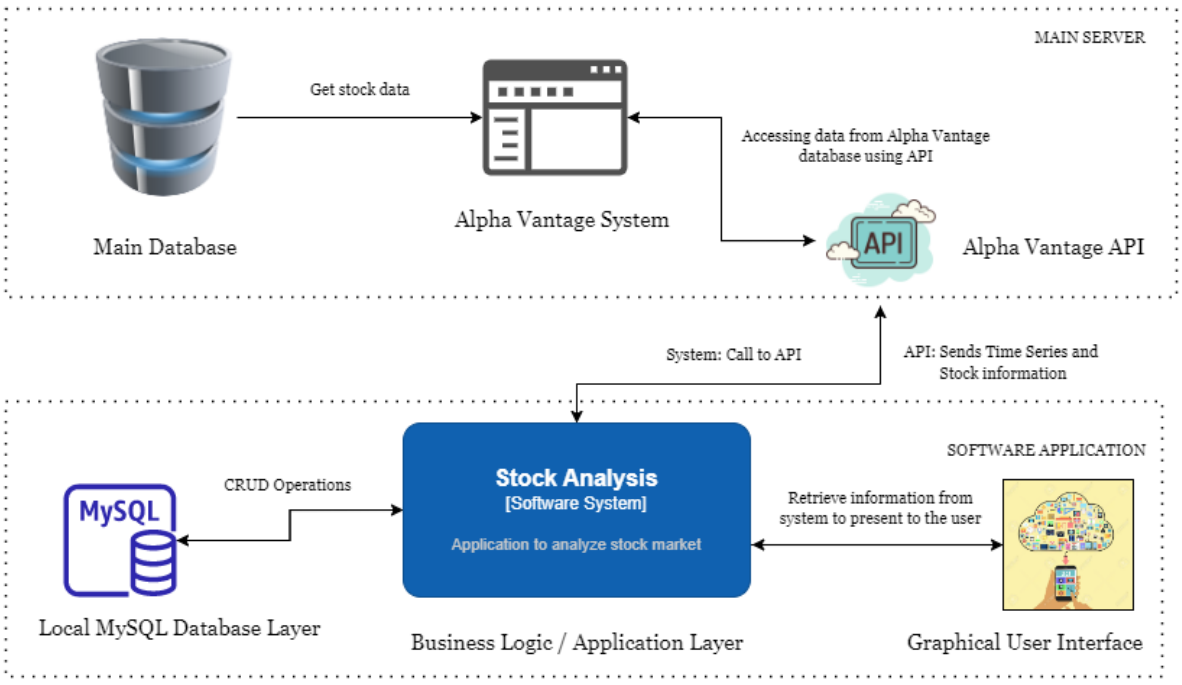
User-friendly: The system should have a shallow learning curve enabling beginners to use the application for analysis of stocks.

Evolvability: The system should readily accept future changes as per changing business requirements.

Scalability: The system should allow scalability to products like forex, cryptocurrency, commodities to a greater extent.

Availability: The system should be available to the user at any given point in time. Scheduled maintenance should be done during the market closing hours.

3. SOFTWARE ARCHITECTURE SUMMARY



The software architecture summary represents the structure of the system as per its implementation. The application can be majorly divided into 2 sections:

The main system, which is the server side of the application, belongs to the API provider Alpha Vantage. The Alpha Vantage system monitors the real time values of its products and performs any CRUD operations necessary for the storage and retrieval of data from the main database. They also provide an API which is free to use with certain restrictions on the number of calls that can be made. This API, communicates with the system and reads the information from the main database of the provider and relays the data in the (.json) format to the user application.

The second part of the application in the user system or the application layer of the architecture constituting the business logic of the application. This layer is responsible for sending an API call to retrieve data from Alpha Vantage and then stores this data in the MySQL database for future use, it collects Intraday Data with a 15-minute interval between consecutive data points and then stores it away along with the Open, High, Low, and Close values of the stock in that interval. It then uses this data to further selectively obtain information on the hourly, daily and weekly trends of the data by calculating the High and Low values and choosing the Open and Close values of the stock for that interval and storing this in individual dictionaries with a triple-nested structure.

The format of the data stored is as follows:

```
{
    "GOOGL": {
        "2022-11-11 20:00:00": {
            "1. open": "96.1500",
            "2. high": "96.2000",
            "3. low": "96.0800",
            "4. close": "96.2000",
        },
        …
        …
        …
        …,
        "2022-09-30 04:15:00": {
            "1. open": "98.7900",
            "2. high": "98.7900",
            "3. low": "98.2400",
            "4. close": "98.2400",
        }
    }
}
```

These data obtained for various intervals are stored as instances of the system. And these are then passed on to the GUI as and when necessary. The GUI is developed to provide a simple yet elegant interface for the user to visualize the state of any given stock anytime and at any interval. The GUI retrieves the intraday, hourly, daily and weekly data from the system. At the click of a button, the stock is chosen and a plot representing the stock is created from the data and it is plotted against time.

3.2. CONSTRAINTS AND DECISIONS

During implementation of the project, we came across some programming, design and architectural constraints which needed to be addressed mainly arising due to the limitations on the API. Using the basic free version of the API allowed a maximum of 5 API calls per minute, this restricts us from extracting large amounts of data for each stock due to real-time processing requirement of the system.

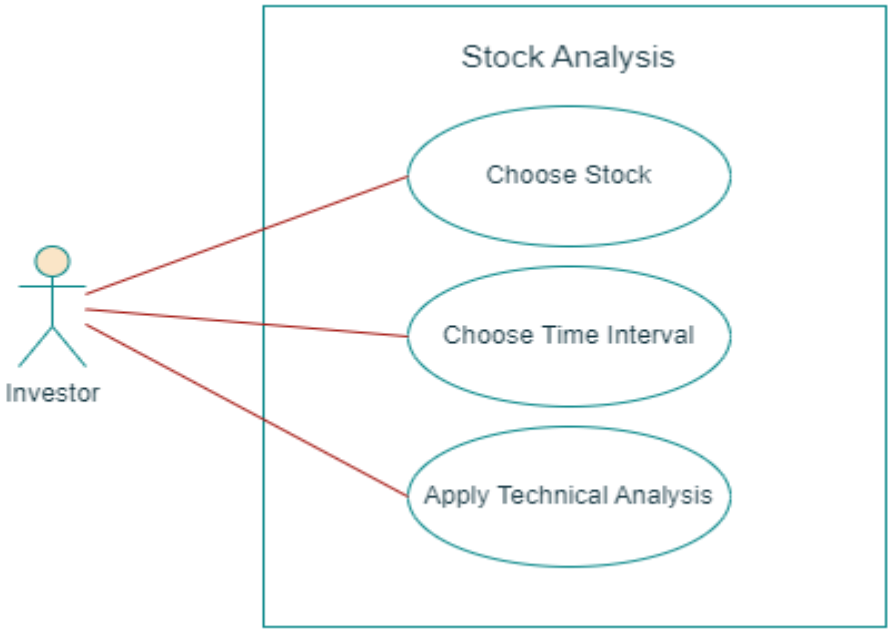
In addition, each stock information would require a minimum of one API call, thus, to continue retrieving the data in real-time, we decided to take a maximum of 5 stocks for our current implementation constraint. However, we still have about 1900 data points for each stock, which is sufficient to display information from a period of 6-7 weeks (~50 days) from the date of enquiry.

Although the Alpha Vantage API provides access to weekly, daily and hourly data, due to this limitation on the API calls we had to alternatively find ways to obtain data, thereby, having to design functionalities to compute the intraday data to retrieve the others.

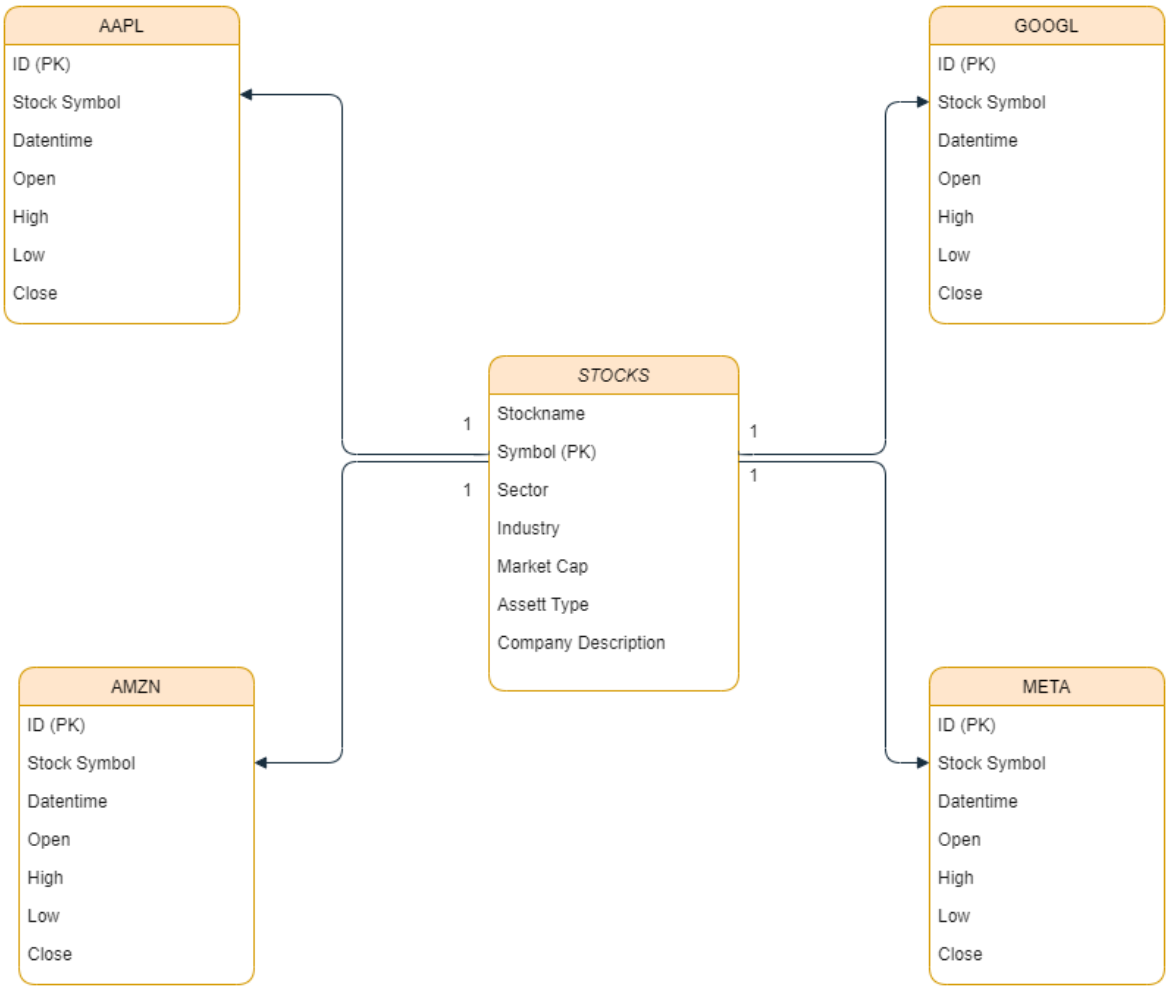
The storage of this data was another concern as storing it in the database directly would lead to creation of either sparse dataset or lead to issues with the database hierarchy due to creation of multiple tables for each stock, thereby leading to compromise in **Space Complexity** of the system. Thus, to reduce the space complexity but not compromising on the time complexity we created dictionaries to temporarily store hourly, daily and weekly data through a single run of the application. Another option would be to store them in the

Python's GUI library TKinter does not support webpage integration i.e., interaction with the graphs in obtaining the high, low and open and close values, would not be possible as in the case when considering a (.webp) format of data. To produce the plot onto the GUI, we save the plots generated into a folder, and retrieve them to display as a (.png) image on a Tkinter.Label field. The denies the possibility of an interaction with the graph.

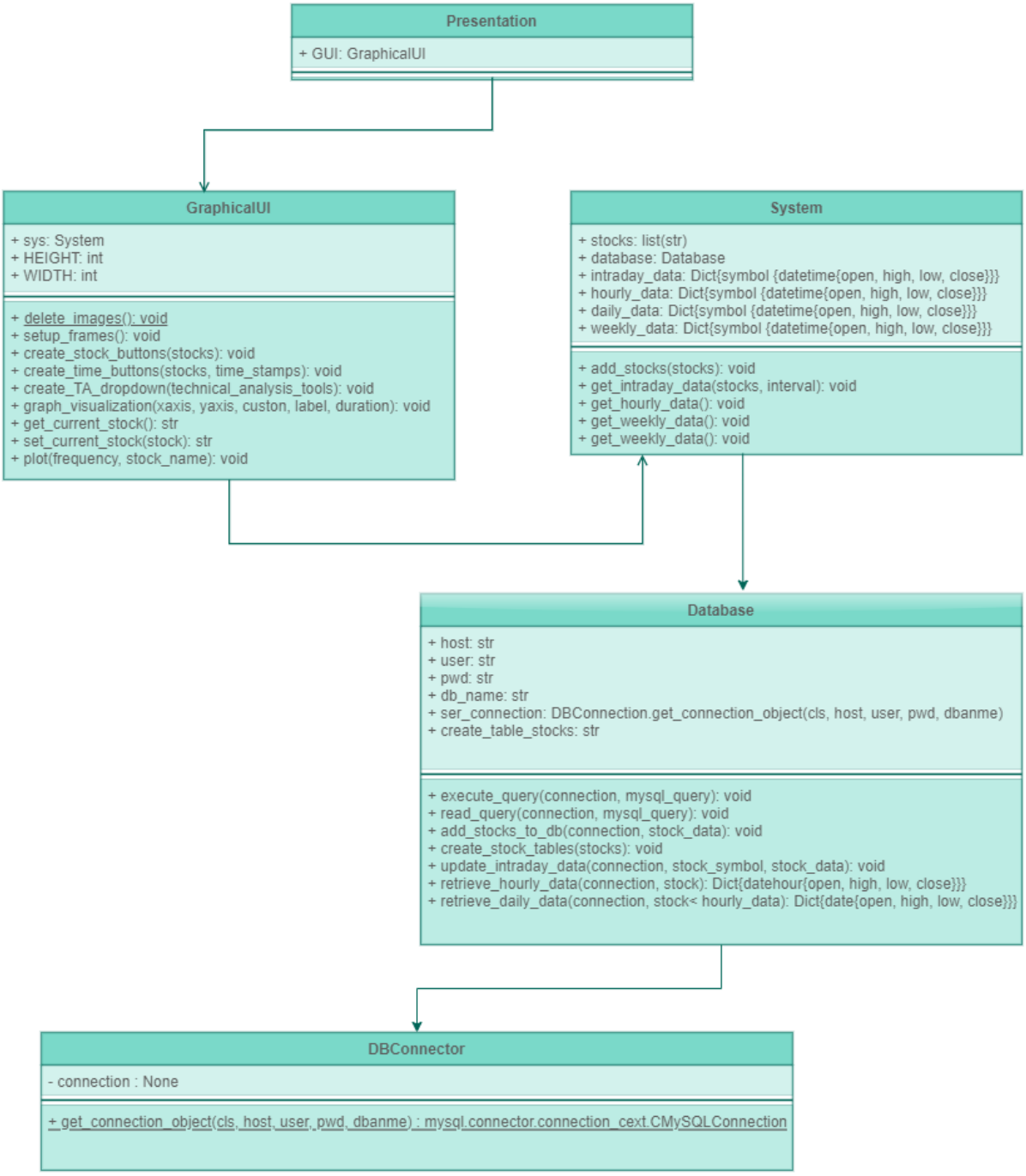
3.3 USE CASE DIAGRAM



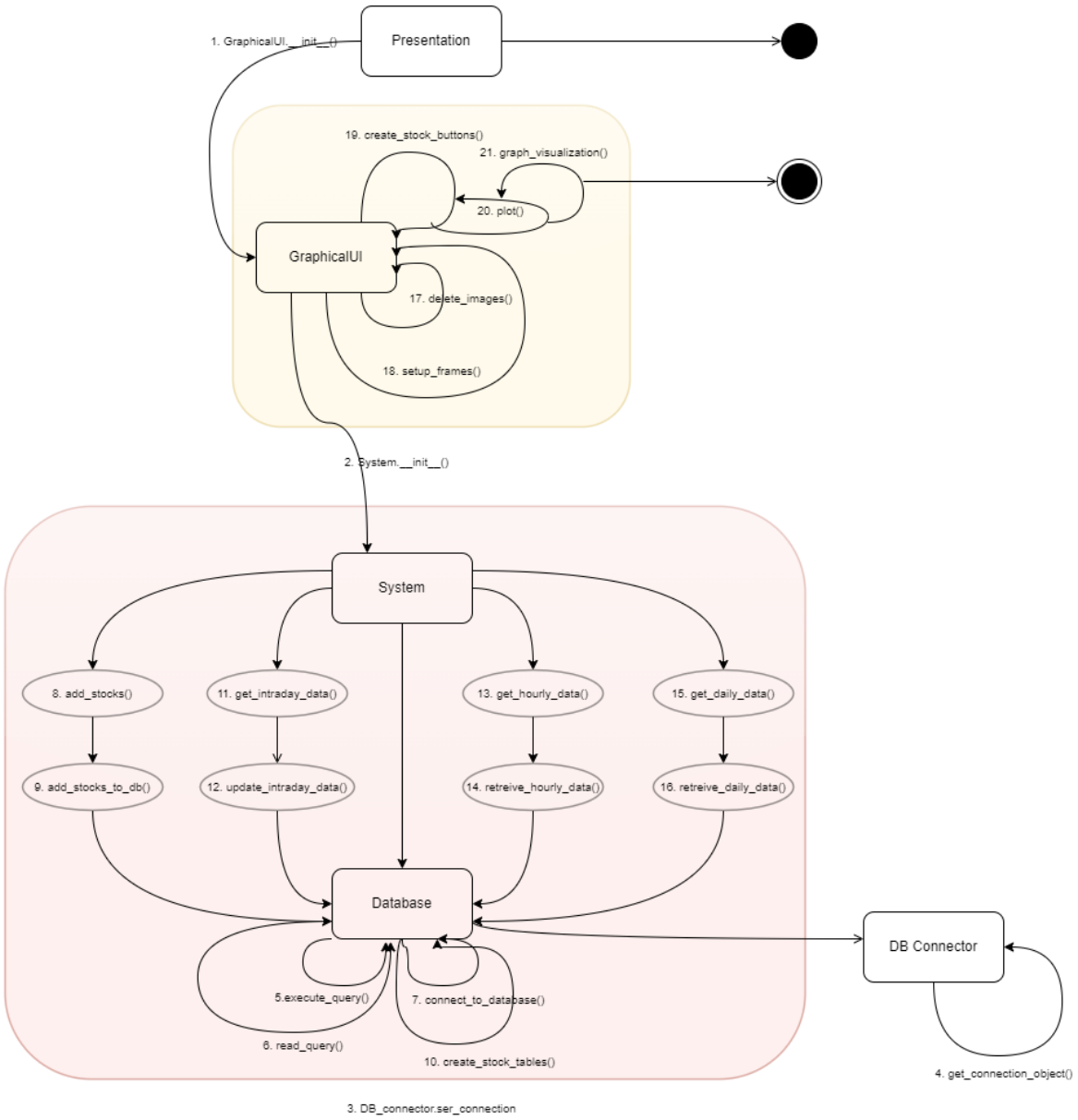
3.4. DATABASE DESIGN



3.5. UML CLASS DIAGRAM



3.6. STATE DIAGRAM



The above figure represents the sequence of events that occur in the application. The Presentation layer creates an object of class GraphicalUI, which is called the default constructor of class System. It executes all methods and assigns states to all the variables of the class. The System class having visibility over the Database class creates an object of the class however, the DB_connector checks whether an object of class Datab ase is already present, if so, it denies object creation due to the Singleton Design Pattern of the system. The DB_connector also checks if the database specified by the user already exists, if the database doesn't exist then the get_connection_object() method creates a new database and

returns it to the user. After the object creation is done, a connection to the database is created which is then used to perform all CRUD operations on the database.

After the connection is established and all read and write queries are ready, we proceed to set-up our database by executing the create_table_stocks query and then design tables for individual stocks at the database level from the create_stocks_tables() method. Then on passing the add_stocks() method in 'sys' which is an instance of class System. The function calls the API for stocks and retrieves information from the API and passes them over to the add_stocks_to_db() function present in 'database' an instance of the Database class. This function contains a MySQL query to create a table called STOCKS and adds important information about certain stocks like Stock Name, Stock Description, Market Value and more. The control is then passed back to the System class where it extracts the intraday values from the API for all the stocks and passes them one at a time from the get_intraday_data() function to the update_intraday_data() function. Simultaneously the hourly, daily and weekly data is also obtained and stored in their respective dictionaries.

Once the data is completely obtained and computations are performed, the control is then passed over from the System class to the GraphicalUI class wherein the system begins to set up the GUI for visualizations. We begin by setting up the canvas using Tkinter and add frames to provide more flexibility to the application window. The buttons are then created for each of the stocks by a call to the method create_stock_buttons(), we also define functionalities to create button for changing the time intervals along with a dropdown menu to choose the Technical Analysis tool.

To plot the graph, we extract the data from the System class and then supply it to the plot() method, which prepares the data into a format ready for plotting. This formatted data is now passed as an argument to the method graph_visualization() which uses python's plotly library to produce proper figures for each stock with up-to-date values. Once the graphs are produced, we save the graphs as (.png) images which are then placed on the GUI in a Tkinter.Label element. When the stock chosen changes the time interval reverts to intraday to show a larger picture before getting into the specifics.

4.1. FUTURE SCOPE

The future scope of the application would involve the addition of Technical Analysis tools to help the user predict the market and invest to get a higher profit-to-loss ratio. We would also focus on improving upon the GUI of the application to provide a better user experience. Interactive graphs would also provide an easier understanding of the charts, enabling the users to study the market better. inclusion of many more stocks along with a variety of products including and not limited to commodities, cryptocurrency and forex.

REFACTORING

1. Composing repetitive statements with a more generalized one:

```
#BEFORE:
button1 = tk.button(self.stock_frame, bd = 1, relief = 'ridge', text = 'GOOGL',
command = partial(self.plot, '15 min', 'GOOGL'))
button1.place(relx = 0.2, rely = 0.2, relwidth = 0.6, relheight = 0.1)
button2 = tk.button(self.stock_frame, bd = 1, relief = 'ridge', text = 'META',
command = partial(self.plot, '15 min', 'META'))
button2.place(relx = 0.2, rely = 0.45, relwidth = 0.6, relheight = 0.1)
button3 = tk.button(self.stock_frame, bd = 1, relief = 'ridge', text = 'AAPL',
command = partial(self.plot, '15 min', 'AAPL'))
button3.place(relx = 0.2, rely = 0.65, relwidth = 0.6, relheight = 0.1)
button4 = tk.button(self.stock_frame, bd = 1, relief = 'ridge', text = 'AMZN',
command = partial(self.plot, '15 min', 'AMZN'))
button4.place(relx = 0.2, rely = 0.8, relwidth = 0.6, relheight = 0.1)
```

```
#AFTER:
def create_stock_buttons(self, stocks):
        for i, j in zip(range(1, len(stocks) + 1), stocks):
            button = tk.Button(self.stock_frame, bd=1, relief='ridge', text="%s" %
(j), command=partial(self.plot, '15 min', j))
            button.place(relx=0.2, rely=0.2 * i, relwidth=0.6, relheight=0.1)
```

2. User Interface Refactoring:

```
#BEFORE:
def create_time_buttons(self, stock, time_stamps=['15 min', '1 H', '1 D', '1 W']):
        for i, j in zip(range(0, len(time_stamps)), time_stamps):
            button = tk.Button(self.time_frame, text="%s" % (j),
command=partial(self.plot, j, stock))
            button.place(relx=0.06 * i * 1.7, rely=0.15)
```

```
#AFTER:
def create_time_buttons(self, stock, time_stamps=['15 min', '1 H', '1 D', '1 W']):
        for i, j in zip(range(0, len(time_stamps)), time_stamps):
            button = tk.Button(self.time_frame, bd=1, relief='ridge', text="%s" %
(j), command=partial(self.plot, j, stock))
            button.place(relx=0.06 * i * 1.7, rely=0.15, relwidth=0.09,
relheight=0.7)
```

3. Refactoring by Abstraction: Pull-Up Method

```
#BEFORE:
def get_daily_data(self, connection, hourly_data):
    for stock in self.stocks:
        cursor = connection.cursor()
        daily_stock_data = {}

        get_dates = """SELECT DISTINCT DATE(DATENTIME) FROM {0} WHERE DATENTIME
LIKE "%20:00:00" """.format(stock)
        cursor.execute(get_dates)
        result = cursor.fetchall()

        for i in range(len(result) - 1):
            open_value = self.read_query(connection,
                                         """SELECT CLOSE FROM {0} WHERE DATENTIME
LIKE "{1}_20:00:00" """.format(stock, result[i + 1][0]))[0][0]
            close_value = self.read_query(connection,
                                         """SELECT CLOSE FROM {0} WHERE DATENTIME
LIKE "{1}_20:00:00" """.format(stock, result[i][0]))[0][0]
            high = [hourly_data[stock][val]['High'] for val in hourly_data[stock]
if(str(result[i][0]) == str(val).split()[0])]
            low = [hourly_data[stock][val]['Low'] for val in hourly_data[stock]
if(str(result[i][0]) == str(val).split()[0])]

            daily_stock_data[str(result[i][0])] = {"Open": open_value, "High":
max(high), "Low": min(low), "Close": close_value}

        daily_data[stock] = daily_stock_data
    return daily_stock_data
```

```
#AFTER:
#In Class System
def get_daily_data(self, connection):
        for stock in self.stocks:
            self.daily_data[stock] = self.database.retrieve_daily_data(connection,
stock, self.hourly_data)
```

```
#AFTER:
#In Class Database
def get_daily_data(self, connection, stock, hourly_data):

        cursor = connection.cursor()
        daily_stock_data = {}

        get_dates = """SELECT DISTINCT DATE(DATENTIME) FROM {0} WHERE DATENTIME
LIKE "%20:00:00" """.format(stock)
        cursor.execute(get_dates)
        result = cursor.fetchall()

        for i in range(len(result) - 1):
```

```
            open_value = self.read_query(connection,
                                         """SELECT CLOSE FROM {0} WHERE DATENTIME
LIKE "{1}_20:00:00" """.format(stock, result[i + 1][0]))[0][0]
            close_value = self.read_query(connection,
                                         """SELECT CLOSE FROM {0} WHERE DATENTIME
LIKE "{1}_20:00:00" """.format(stock, result[i][0]))[0][0]
            high = [hourly_data[stock][val]['High'] for val in hourly_data[stock]
if(str(result[i][0]) == str(val).split()[0])]
            low = [hourly_data[stock][val]['Low'] for val in hourly_data[stock]
if(str(result[i][0]) == str(val).split()[0])]

            daily_stock_data[str(result[i][0])] = {"Open": open_value, "High":
max(high), "Low": min(low), "Close": close_value}

        return daily_stock_data
```

4. Composing methods by extracting and creating abstraction using Pull-up Method:

```
#BEFORE:

def get_values(stock, interval, data):
    intraday_values = []
    for datetime in data:
        intraday_values.append([datetime, float(data[datetime]['1. open']),
float(data[datetime]['2. high']), float(data[datetime]['3. low']),
float(data[datetime]['4. close'])])
    return (intraday_values)


def update_intraday_values(stocks, interval = '15min'):
    for stock in stocks:
        execute_query(connection, """DELETE FROM %s""" %(stock))
        url =
'https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&interval=%s&symbol
=%s&outputsize=full&apikey=7NVAEAZYAXPYKMZB' %(interval, stock)
        intraday_data = requests.get(url).json()
        values = get_values(stock, interval, intraday_data['Time Series (%s)'
%(interval)])
        for n, i in zip(range(1, len(values) + 1), values):
            intraday_update_query= """INSERT INTO %s VALUES (%s, '%s', '%s', %f,
%f, %f, %f)""" % (stock, n, intraday_data['Meta Data']['2. Symbol'], i[0], i[1],
i[2], i[3], i[4])
            execute_query(connection, intraday_update_query)

def get_intraday_data(connection, stocks):
    for stock in stocks:
        intraday_data[stock] = {}
        query = """SELECT * FROM %s"""%(stock)
        val = read_query(connection, query)
        for i in val:
            intraday_data[stock][str(i[2])] = {'Open': i[3], 'High': i[4], 'Low':
i[5], 'Close': i[6]}

    return intraday_data
```

```
#AFTER:
#In Class System
def get_intraday_data(self, stocks, interval='15min'):
    print("Refreshing the database with the latest data. Please wait while we load
visualizations for you...")
    for stock in stocks:
        values = []
        self.intraday_data[stock] = {}
        url =
'https://www.alphavantage.co/query?function=TIME_SERIES_INTRADAY&interval=%s&symbol
=%s&outputsize' \
              '=full&apikey=7NVAEAZYAXPYKMZB' % (
                  interval, stock)
        intraday = requests.get(url).json()
        data = intraday['Time Series (%s)' % interval]
        for datetime in data:
            values.append([datetime, float(data[datetime]['1. open']),
float(data[datetime]['2. high']),
                           float(data[datetime]['3. low']),
float(data[datetime]['4. close'])])
            self.intraday_data[stock][datetime] = {'Open': float(data[datetime]['1.
open']), 'High': float(data[datetime]['2. high']), 'Low': float(data[datetime]['3.
low']), 'Close': float(data[datetime]['4. close'])}
        self.database.update_intraday_data(self.database.connection, stock, values)
```

```
#In Class Database:
def update_intraday_data(self, connection, stock_symbol, stock_data):
    self.execute_query(connection, "TRUNCATE TABLE %s" % stock_symbol)
    for num, item in zip(range(1, len(stock_data) + 1), stock_data):
        intraday_update_query = """INSERT INTO %s VALUES (%s, '%s', '%s', %f, %f,
%f, %f)""" % (
            stock_symbol, num, stock_symbol, item[0], item[1], item[2], item[3],
item[4])
        self.execute_query(connection, intraday_update_query)
```

5. Moving features between objects:

```
#Before:
#Creating stocks: list(str) in each class in the application
```

```
#AFTER:
self.sys = System.System()
self.create_stock_buttons(self.sys.stocks)
```

6. Extract Class Refactoring:

```
#BEFORE:
"""All functionalities were a part of a single class which dealt with computation,
storage, retrieval and display of information."""
```

```
#AFTER:
"""The functionalities were separated into classes and a heirarchy was created to
ensure security and abstraction of data.
The Layers include Presentation, GraphicalUI, System, Database and DB connector"""
```

GITHUB REPOSITORY

All the project files including diagrams, code, images and report are available at GitHub repository

CONCLUSION

The Stock Analysis and Visualization application, developed as part of the SOEN6441 Fall 2022 Project requirement has been a wonderful learning experience. The project provided the necessary exposure to build software applications from the ground-up.

This project deepened our understanding of Python programming constructs with the use of good coding standards, and gave us a great deal of hands-on experience with various python packages like my-sql-connector, pyplot, requests and tkinter.

We learnt various Design Constraints and Patterns which are crucial to development and maintenance of good software. Testing the functionality of the application in increments and refactoring the code to ensure ease of understanding were the most important phases of our project development. The thrill of tackling and overcoming various design and development challenges gave us immense joy and a deeper understanding of the industry standards for Application Development.