# Data Engineering Principles and Practice Final Project

By

Bo-Mi Kim, Cody Kim, Greyson Mouser, and Michael Paisner

# Table of Contents

# Stock Information Relational Database and API Application

## Dataset Fetching

Three API sources were used for fetching the data for our database: Yahoo Finance, newsapi.org, and Financial Modeling Prep. Each of these sources uses the requests module in Python to create the request and store the result and the result is transformed from that into a state where its ready to be inserted into the PostgreSQL database that was set up.

### *Stock Information Table*

To gather the stock prices at open, close, lowest and highest price each day, and total volume traded, we used a call to the Yahoo Finance API (yfinance). This API makes a call to the Yahoo Finance website and returns a data frame that is based on the parameters we passed.

```
# Define global variables
stocklist =
["PFE","MRK","JNJ","AMGN","BIIB","GILD","MDT","ABT","BDX","UNH","HCA","HUM"]

# Define required variables:
data_needed = ['Open', 'Adj Close', 'Low', 'High', 'Volume']

# Get start date and convert to string
start = get_start_date().strftime('%Y-%m-%d')

# Get end date and convert to string
end = (get_end_date() + timedelta(days=1)).strftime('%Y-%m-%d')
```

Note: get_start_date and get_end_date functions are described in the Automation and API section.

Once we settled on a date range of information required, the data is placed into variables that will be used to scrape the Yahoo Finance API. We created a list of the stocks that were identified in our project proposal with ties to the biomedical/tech industry. Before sending the stocks to the website to pull the data, we defined the parameters, and used a for loop to have the data frame build upon itself by specific ticker. This is shown below:

```
# For each stock in the stocklist, get stock information from yfinance
for stock in stocklist:
      # Define the parameters for yfinance
      y_params = {
```

```
            'tickers': stock,
            'start': start,
            'end': end,
            'interval': '1d'
      }

      # Create the blank dataframe
      stocks = pd.DataFrame()

      # For each stock, get the information and add to the stocks dataframe
      for x in data_needed:
            data = yf.download(**y_params)[x]

            stocks[x] = data

      stocks['stock'] = stock

      stocks['date'] = data.index

      if firstpass:
            final_stocks = stocks.copy()
            firstpass = False
      else:
            final_stocks = pd.concat([final_stocks, stocks], ignore_index=True,
axis=0)

      del(stocks)
```

Because the data is already cleaned, there were no additional steps required for data transformations or feature manipulation. This is shown by printing the head of the data frame:

```
final_stocks.head()
```

| | Open | Adj Close | Low | High | Volume | stock | date |
|---|---|---|---|---|---|---|---|
| 0 | 27.610001 | 27.590000 | 27.520000 | 28.139999 | 34232000 | PFE | 2024-02-20 |
| 1 | 27.600000 | 27.670000 | 27.360001 | 27.680000 | 27370600 | PFE | 2024-02-21 |
| 2 | 27.590000 | 27.549999 | 27.190001 | 27.700001 | 31957500 | PFE | 2024-02-22 |
| 3 | 27.750000 | 27.760000 | 27.690001 | 28.090000 | 33182600 | PFE | 2024-02-23 |
| 4 | 27.670000 | 27.180000 | 27.070000 | 27.670000 | 45685500 | PFE | 2024-02-26 |

The data is then sent to the database for use within our pipeline.

```
##### export stocks to PostgreSQL
        conn = establish_db_conn()
```

```
        cursor = conn.cursor()

        cursor.execute(
            '''CREATE TABLE IF NOT EXISTS stockinfo.stockinfo
            (
            symbol VARCHAR(3) NOT NULL,
            info_date DATE NOT NULL,
            open_price REAL,
            close_price REAL,
            daily_low_price REAL,
            daily_high_price REAL,
            volume BIGINT,
            PRIMARY KEY(symbol, info_date)
            );
            '''
        )
        conn.commit()

        for index, row in final_stocks.iterrows():
            query = '''INSERT INTO stockinfo.stockinfo
                (
                symbol,
                info_date,
                open_price,
                close_price,
                daily_low_price,
                daily_high_price,
                volume
                )
                VALUES(
                %s,
                %s,
                %s,
                %s,
                %s,
                %s,
                %s
                )
                ON CONFLICT (symbol, info_date) DO NOTHING;'''
            args = (
                row["stock"],
                str(row["date"])[:10],
                row["Open"],
                row["Adj Close"],
                row["Low"],
```

```
                row["High"],
                row["Volume"],
            )
            cursor.execute(query, args)
            conn.commit()
        conn.close()
```

Once committed to the database, our database updates to look like the following:

| symbol [PK] character varying (3) | info_date [PK] date | open_price real | close_price real | daily_low_price real | daily_high_price real | volume bigint |
|---|---|---|---|---|---|---|
| PFE | 2024-04-15 | 25.91 | 25.91 | 25.75 | 26.17 | 35660200 |
| PFE | 2024-04-16 | 25.82 | 25.69 | 25.68 | 25.99 | 28885300 |
| PFE | 2024-04-17 | 25.69 | 25.42 | 25.26 | 25.69 | 39810500 |
| MRK | 2024-04-15 | 126.69 | 126.19 | 125.87 | 127.82 | 6241000 |
| MRK | 2024-04-16 | 126.56 | 125.06 | 125.03 | 126.8 | 5587300 |
| MRK | 2024-04-17 | 125.67 | 125.37 | 124.71 | 126.19 | 4771584 |

Where the stock symbol and date are primary keys in which will enable the user to join the news data and the rating data together as can be seen in the ERD in a later section.

### *Stock News Dataset*

The second dataset we created contains the 100 most relevant news articles related to 12 companies in the healthcare and pharmaceutical industries: Pfizer, Merck, Johnson & Johnson, Amgen, Biogen, Gilead, Medtronic, Abbott Laboratories, Becton, Dickinson and Co, UnitedHealth Group, HCA Healthcare, Humana. We make an API request to newsapi.org for each company and append each article to the end of our dataframe:

```
# request for each company and append articles to the list
for company, ticker in companies:
      params = {
            'q': company,
             'apiKey': api_key
      }
      response = requests.get(url, params=params, verify=False)
      if response.status_code == 200:
            articles = response.json()['articles']
            for article in articles:
                  data.append({
                        'company': company,
                        'ticker': ticker,
                        'title': article['title'],
                        'description': article.get('description', ''),
```

```
                        'author': article.get('author', ''),
                        'date': article.get('publishedAt', '')[:10],
                        'url': article['url']
                    })
      else:
            raise ValueError(f'Failed to fetch news for {company}:',
response.status_code, response.text)

        # Create a dataframe from the news information
        news = pd.DataFrame(data)
```

The result is a dataset that is in a suitable format for our database that includes the following columns:

symbol: The ticker symbol of the company.

title: The title of the news article.

description: A short description of the news article.

author: The author of the news article.

article_date: The date the news article was published.

url: The URL of the news article.

### *Stock Ratings and Recommendations*

To get the recommendations for stocks, we are using the Financial Modeling Group's API service. The query being run here is to get historical ratings for each stock in the stocklist. The information is received via the request to the API service and the results are sorted into a pandas dataframe:

```
# Create or append to the dataframe the stock recommendations by stock and by day
if response.status_code == 200:
      if firstpass:
            ratings = pd.DataFrame(response.json())
            ratings = ratings[(pd.to_datetime(ratings.date) >
pd.to_datetime(get_start_date())) & (pd.to_datetime(ratings.date) <=
pd.to_datetime(get_end_date()))]
            firstpass = False
      else:
            temp = pd.DataFrame(response.json())
            temp = temp[(pd.to_datetime(temp.date) >
pd.to_datetime(get_start_date())) & (pd.to_datetime(temp.date) <=
pd.to_datetime(get_end_date()))]
```

```
            ratings = pd.concat([ratings,temp], axis=0).reset_index(drop=True)
            del temp
else:
      raise ValueError(f'Failed to fetch recommendations for {stock}:
{response.status_code}, {response.text}')
```

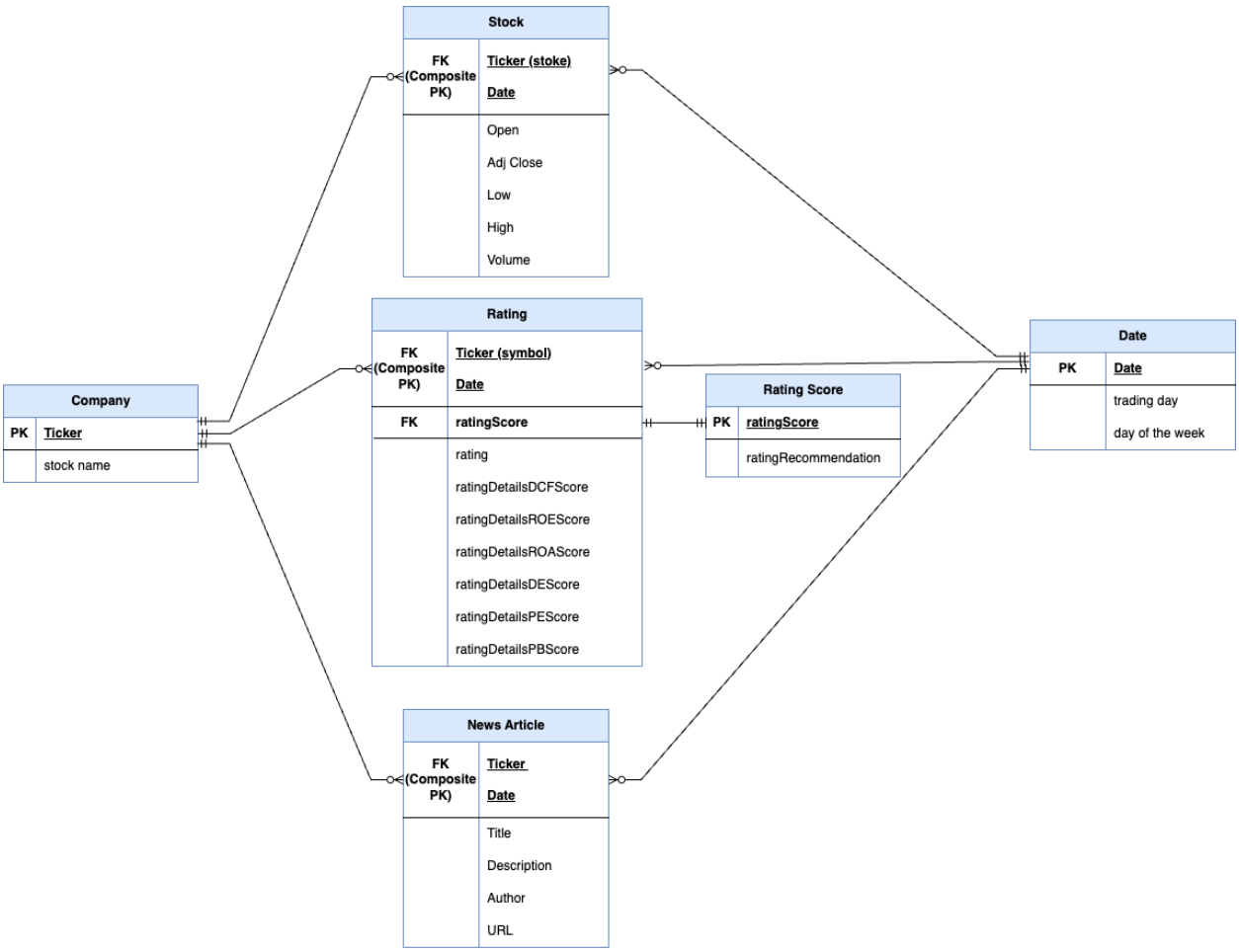
The ratings dataframe is then sorted and the integer mapping to the rating (sell, buy, neutral, etc.) is extracted from the information is the ratings table. Both are then uploaded to the SQL server using the database connection.

### *Loading data into the database*

For loading data into a database using Python, we initially considered SQLite due to its lightweight nature and ease of use. However, we ultimately chose PostgreSQL for its better compatibility with Apache Airflow. PostgreSQL offers robust features and scalability, making it well-suited for our project's workload and use case. Below is a the custom developed function showing how we connected to the PostgreSQL database throughout the code:

```
# Create a function to establish a database connection to PostgreSQL
def establish_db_conn():
      conn = psycopg2.connect(dbname='jhu')
      return conn
```

## *Entity-Relationship Diagram (ERD) of our schema*




The **Company** entity is central to the schema, representing businesses listed on the stock market. It contains details such as the stock name and its unique stock ticker symbol, the latter of which is the primary key for this entity. This primary key uniquely identifies each company in the database and is critical in linking company-related data across the schema.

Moving on to the **Stock** entity, this contains daily trading data for stocks, with fields for the opening price, adjusted closing price, daily low and high prices, and trading volume. The primary key is a combination of the stock ticker and the date, creating a unique identifier for each stock entry on a given date. The stock ticker here serves as a foreign key, referencing the primary key of the Company entity. This establishes a one-to-many relationship between a Company and its Stock records since a company can have multiple stock entries over different dates.

The **NewsArticle** entity captures information about news articles related to companies, with attributes for the article's title, description, author, publication date, and the URL where the article can be accessed. Like the Stock entity, it employs a composite primary key comprising the ticker and the date, assuming there's at most one article per company per day. The ticker symbol is used as a foreign key that links back to the Company entity, demonstrating a one-to-many relationship, as a company can be associated with multiple news articles over time.

The **Rating** entity holds financial ratings for companies, with various score attributes and the primary key consisting of the ticker and date, ensuring each rating is distinct for a company on a given date. The relationship between Company and Rating is one-to-many, facilitated by the ticker symbol in the Rating entity, which points to the Company entity's primary key. Additionally, the 'RatingScore' acts as a foreign key within the Rating entity, establishing a linkage to the RatingScore entity.

The **RatingScore** entity is an addition meant to encapsulate a general assessment of a company's financial rating. It holds two attributes: a numeric score and a text-based recommendation such as "strong", "medium", or "low". The RatingScore is its primary key and is linked to the Rating entity through a one-to-one relationship. Each financial rating entry is directly connected to one RatingScore, which provides an overview or recommendation based on the detailed scores.

Lastly, the **Date** entity is introduced to manage the dates within the database. Its single attribute, Date, is also its primary key. This entity is pivotal as it forms one-to-many relationships with the Stock, NewsArticle, and Rating entities. Each date can correspond to several records within these entities, meaning a particular date may see multiple stocks traded, news articles published, and ratings assigned, across different companies.

In summary, the schema is designed to efficiently interlink various aspects of financial data concerning companies. The primary keys ensure uniqueness within each entity, while foreign keys establish relationships that allow the dataset to be queried in a multifaceted way, providing comprehensive insights into company performance, news sentiment, and financial ratings.

## Automation and API

To create the dataflow and the ability to query the information, we used Python's Airflow package in conjunction with the 'jhu' PostgreSQL database of the class-provided Ubuntu

VM and Python's Flask application to create a REST API. See attached documentation for Python code.

### *Automation with Airflow*

The Airflow application works through seven operators: 6 PythonOperators and 1 DummyOperator. Our Airflow is started by running the "conda activate airflow_env" then running "airflow webserver -p 8080" as commands in the terminal. The scheduler is then started by running the "conda activate airflow_env" and "airflow scheduler" commands in a new terminal window.

The workflow starts with the "check_db_health" task by creating the "stockinfo" schema of the "jhu" database if that schema does not already exist. Without this precaution, if the workflow had not been run on the VM before, all following SQL queries would fail as no "stockinfo" schema would exist. The operator creates the schema by using a user-defined function to connect to the PostgreSQL database with the psycopg2 Python package

```
def establish_db_conn():
      conn = psycopg2.connect(dbname='jhu')
      return conn
```

and running the SQL CREATE TABLE query using the corresponding cursor:

```
# This function just creates the schema if the schema doesn't currently exist in
postgresql
def check_db_health():
      conn = establish_db_conn()
      cursor = conn.cursor()

      # Create schema
      cursor.execute('''CREATE SCHEMA IF NOT EXISTS stockinfo''')
      conn.commit()
      conn.close()
```

A DummyOperator (no action is completed here) then allows the workflow to branch to four different Python operators. Each of these operators create data tables in our SQL database.

The four operators are: "load_stock_recs", "get_stock_info", "get_stock_news", and "get_stock_names". The load_stock_recs, get_stock_info, and get_stock_news pulls information as described above in the Dataset Fetching section. The get_stock_names operator retrieves the stocks' names through a pre-defined list of tuples (with both the stock tickers and names) that then get loaded into the database.

Lastly, the four operators merge on a “get_date_table” task. This operator uses the same API service as the load_stock_recs task to pull information on when the stock market is opened and closed. The operator will then loop from the start date to the end date of the data pull for each day and record whether the market was open or closed, based on if it was a weekend or if there was a holiday.

The start and end dates of the data pull are also determined based on user defined functions. The start date will be either the last date available in the stockinfo.dates table if the table exists or it will be set to 30 days prior to the current date. This was done so that we only need to query information for days that we do not have data for. If the stockinfo.dates table is not currently in the schema, it sets the start date to 30 days back - the furthest out all our API sources are able to go. The end date function returns the most recently completed day. These functions are used throughout the automation to retrieve a consistent start/end date.

### *Retrieving Data with the API*

Along with the Airflow automation is a REST API that can be run to retrieve the information stored in the PostgreSQL database. As mentioned, the API runs in a Python script with the Flask application.

For this implementation, the flask application only has a GET task in a function called “query”. Data should not be added or removed from this database as all data changes/updates are handled in the Airflow workflow - the only functionality is to retrieve data from the current tables. The API runs on the localhost at port 8080 and must be started using the conda airflow_env. The API starts by running python script “stockinfo_flask.py”. Once this is running the user can use the API URL string “http://localhost:8001/api/query?table=stockinfo.[insert table name]” to run a query without any filters on one of the tables.

To get specific, filtered information, three parameters have also been set up: stock, startdate, and enddate. If any other parameter is entered an error message will be returned stating to give a valid parameter. To add a parameter, you would add “&param=[param value]” to the end of the URL. For example, to retrieve the information for only Pfizer between April 7, 2024 and April 10, 2024, the URL string would look like this:

```
http://localhost:8001/api/query?table=stockinfo.stockinfo&stock='PFE'&startdate='
2024-04-07'&enddate='2024-04-10'
```

Please note that dates must be in ‘YYYY-MM-DD’ format.

The limitation on the filters is that only one parameter value can be given at a time. For example, you cannot give two stocks to search either by using an additional "&stock='MRK'" param or by giving two parameter values like this "&stock=['MRK','PFE']" or in any other format.

Another limitation to note is that the API returns the date in time since epoch in milliseconds. To get the correct date, please convert the date column from time since epoch to date format using any method of your choosing.

Please see the following API URL request examples:

*To get all stockinfo.ratings table information:*

```
url = 'http://localhost:8001/api/query?table=stockinfo.ratings'
```

*To get all stockinfo.stockinfo information for Pfizer:*

```
url = 'http://localhost:8001/api/query'?table=stockinfo.stockinfo&stock='PFE'
```

To get all stockinfo.stokinfo information for after April 16, 2024:

```
url =
'http://localhost:8001/api/query'?table=stockinfo.stockinfo&startdate='2024-04-
16'
```

# Attachments

## Airflow Application and DAGs

```
# import required modules:
import os
import json
from airflow import DAG
from airflow.operators.python_operator import PythonOperator, BranchPythonOperator
from airflow.operators.dummy_operator import DummyOperator
from datetime import datetime, timedelta, date
import requests
import pandas as pd
import yfinance as yf
import psycopg2

# define the default arguments:
default_args = {
    'owner': 'data_engineer',
    'start_date': datetime(2024, 4, 2),
    'retries': 1,
    'retry_delay': timedelta(seconds=30)
}

# Define global variables
stocklist = ["PFE","AMGN","MRK","JNJ","BIIB","GILD","MDT","ABT","BDX","UNH","HCA","HUM"]

# Define the DAG
with DAG('get_stock_data', default_args=default_args, schedule_interval='@daily') as dag:

    # Create a function to establish a database connection to postgresql
    def establish_db_conn():
        conn = psycopg2.connect(dbname='jhu')
        return conn

    # Create a function to get the start date,
    # either 30 days back or from the previous run date (
    # which is based on the dates table)
    def get_start_date():
        conn = establish_db_conn()
        cursor = conn.cursor()

        # Get the startdate from stockinfo.dates, otherwise
        # use 30 days ago
```

```
        try:
            cursor.execute('''SELECT * FROM stockinfo.dates ORDER BY market_date DESC''')
            startdate = cursor.fetchall()[0][0]
            startdate = datetime.strptime(startdate, '%Y-%m-%d').date()
            return startdate
        except:
            return date.today() - timedelta(days=30)

    # Create a function to get the end date, which will always be the
    # day before the current date (to prevent partial days being uploaded to the DB)
    def get_end_date():
        return date.today() - timedelta(days=1)

    # This function just creates the schema if the schema doesn't currently exist in
postgresql
    def check_db_health():
        conn = establish_db_conn()
        cursor = conn.cursor()

        # Create schema
        cursor.execute('''CREATE SCHEMA IF NOT EXISTS stockinfo''')
        conn.commit()
        conn.close()

    # Create a function to load the stock recommendations table
    def load_stock_recs():

        # Define api_key
        api_key = "apikey=5036b66d0f6fd3c5eb559e1c2abade9b"

        # For firstpass, we will want to create the dataframe, otherwise we will append
        firstpass = True

        # For each stock, complete the API request for the specific stock and then
create/append to the ratings dataframe.
        for stock in stocklist:
            api_request = f"https://financialmodelingprep.com/api/v3/historical-
rating/{stock}?{api_key}"
            response = requests.get(api_request)

            # Create or append to the dataframe the stock recommendations by stock and by day
            if response.status_code == 200:
                if firstpass:
                    ratings = pd.DataFrame(response.json())
```

```
                    ratings = ratings[(pd.to_datetime(ratings.date) >
pd.to_datetime(get_start_date())) & (pd.to_datetime(ratings.date) <=
pd.to_datetime(get_end_date()))]
                    firstpass = False
                else:
                    temp = pd.DataFrame(response.json())
                    temp = temp[(pd.to_datetime(temp.date) >
pd.to_datetime(get_start_date())) & (pd.to_datetime(temp.date) <=
pd.to_datetime(get_end_date()))]
                    ratings = pd.concat([ratings,temp], axis=0).reset_index(drop=True)
                    del temp
            else:
                raise ValueError(f'Failed to fetch recommendations for {stock}:
{response.status_code}, {response.text}')

        # Sort the ratings dataframe by date and then symbol
        ratings =
ratings.sort_values(by=["date",'symbol'],ascending=False).reset_index(drop=True)

        # Create new table for mapping the ratings scores to the recommendations
        ratingmapping =
ratings[['ratingScore','ratingRecommendation']].drop_duplicates(['ratingScore','ratingRecomme
ndation']).reset_index(drop=True)

        ##########
        ### Create code for putting ratingmapping into SQL
        ##########
        conn = establish_db_conn()
        cursor = conn.cursor()

        cursor.execute('''DROP TABLE IF EXISTS stockinfo.ratingmapping''')
        conn.commit()

        cursor.execute(
            '''CREATE TABLE IF NOT EXISTS stockinfo.ratingmapping
            (
            ratingScore INTEGER PRIMARY KEY,
            ratingRecommendation TEXT
            );'''
        )
        conn.commit()

        for index, row in ratingmapping.iterrows():
            query = '''INSERT INTO stockinfo.ratingmapping (ratingScore,
ratingRecommendation)
```

```
            VALUES (%s, %s)
            ON CONFLICT (ratingScore) DO NOTHING;'''
        args = (str(row["ratingScore"]), row["ratingRecommendation"],)
        cursor.execute(query, args)
        conn.commit()

    # Remove columns from the original table for the Recommendations
    ratings = ratings[[cols for cols in ratings.columns if "Recommendation" not in cols]]

    ##########
    ### Create code for putting ratings into SQL
    ##########
    cursor.execute(
        '''CREATE TABLE IF NOT EXISTS stockinfo.ratings
        (
        symbol VARCHAR(4) NOT NULL,
        rating_date DATE NOT NULL,
        rating TEXT,
        ratingScore INTEGER,
        ratingDetailsDCFScore INTEGER,
        ratingDetailsROEScore INTEGER,
        ratingDetailsROAScore INTEGER,
        ratingDetailsDEScore INTEGER,
        ratingDetailsPEScore INTEGER,
        ratingDetailsPBScore INTEGER,
        PRIMARY KEY(symbol, rating_date)
        );'''
    )
    conn.commit()

    for index, row in ratings.iterrows():
        query = '''INSERT INTO stockinfo.ratings
            (
            symbol,
            rating_date,
            rating,
            ratingScore,
            ratingDetailsDCFScore,
            ratingDetailsROEScore,
            ratingDetailsROAScore,
            ratingDetailsDEScore,
            ratingDetailsPEScore,
            ratingDetailsPBScore
            )
            VALUES(
```

```
            %s,
            %s,
            %s,
            %s,
            %s,
            %s,
            %s,
            %s,
            %s,
            %s
            )
            ON CONFLICT (symbol, rating_date) DO NOTHING;'''

        args = (row["symbol"],
                row["date"],
                row["rating"],
                row["ratingScore"],
                row["ratingDetailsDCFScore"],
                row["ratingDetailsROEScore"],
                row["ratingDetailsROAScore"],
                row["ratingDetailsDEScore"],
                row["ratingDetailsPEScore"],
                row["ratingDetailsPBScore"],)

        cursor.execute(query, args)
        conn.commit()
    conn.close()

def get_stock_info():

    # Define required variables:
    data_needed = ['Open', 'Adj Close', 'Low', 'High', 'Volume']

    # Get start date and convert to string
    start = get_start_date().strftime('%Y-%m-%d')

    # Get end date and convert to string
    end = (get_end_date() + timedelta(days=1)).strftime('%Y-%m-%d')

    # Create first pass variable:
    firstpass = True

    # For each stock in the stocklist, get stock information from yfinance
    for stock in stocklist:
        # Define the parameters for yfinance
```

```
        y_params = {
        'tickers': stock,
        'start': start,
        'end': end,
        'interval': '1d'
        }

        # Create the blank dataframe
        stocks = pd.DataFrame()

        # For each stock, get the information and add to the stocks dataframe
        for x in data_needed:
            data = yf.download(**y_params)[x]

            stocks[x] = data

        stocks['stock'] = stock

        stocks['date'] = data.index

        if firstpass:
            final_stocks = stocks.copy()
            firstpass = False
        else:
            final_stocks = pd.concat([final_stocks, stocks], ignore_index=True, axis=0)

        del(stocks)

    ##### export stocks to PostgreSQL
    conn = establish_db_conn()
    cursor = conn.cursor()

    cursor.execute(
        '''CREATE TABLE IF NOT EXISTS stockinfo.stockinfo
        (
        symbol VARCHAR(4) NOT NULL,
        info_date DATE NOT NULL,
        open_price REAL,
        close_price REAL,
        daily_low_price REAL,
        daily_high_price REAL,
        volume BIGINT,
        PRIMARY KEY(symbol, info_date)
        );
        '''
```

```
        )
        conn.commit()

        for index, row in final_stocks.iterrows():
            query = '''INSERT INTO stockinfo.stockinfo
                (
                symbol,
                info_date,
                open_price,
                close_price,
                daily_low_price,
                daily_high_price,
                volume
                )
                VALUES(
                %s,
                %s,
                %s,
                %s,
                %s,
                %s,
                %s
                )
                ON CONFLICT (symbol, info_date) DO NOTHING;'''
            args = (
                row["stock"],
                str(row["date"])[:10],
                row["Open"],
                row["Adj Close"],
                row["Low"],
                row["High"],
                row["Volume"],
            )
            cursor.execute(query, args)
            conn.commit()
        conn.close()

    def get_stock_news():

        # Define API key
        api_key = 'dec287b1a57c46009b0fa76cdc07f128'

        # List of companies and their ticker symbols
        companies = [
            ('Pfizer', 'PFE'), ('Merck', 'MRK'), ('Johnson & Johnson', 'JNJ'),
```

```
            ('Amgen', 'AMGN'), ('Biogen', 'BIIB'), ('Gilead', 'GILD'),
            ('Medtronic', 'MDT'), ('Abbott Laboratories', 'ABT'), ('Becton, Dickinson and
Co', 'BDX'),
            ('UnitedHealth Group', 'UNH'), ('HCA Healthcare', 'HCA'), ('Humana', 'HUM')
        ]

        # Get endpoint URL and parameters
        url = 'https://newsapi.org/v2/everything'

        # Initialize empty list to store news articles
        data = []

        # request for each company and append articles to the list
        for company, ticker in companies:
            params = {
                'q': company,
                'apiKey': api_key
            }
            response = requests.get(url, params=params, verify=False)
            if response.status_code == 200:
                articles = response.json()['articles']
                for article in articles:
                    data.append({
                        'company': company,
                        'ticker': ticker,
                        'title': article['title'],
                        'description': article.get('description', ''),
                        'author': article.get('author', ''),
                        'date': article.get('publishedAt', '')[:10],
                        'url': article['url']
                    })
            else:
                raise ValueError(f'Failed to fetch news for {company}:',
response.status_code, response.text)

        # Create a dataframe from the news information
        news = pd.DataFrame(data)
        # Remove null article titles, replacing with NOT NULL due to errors.
        news = news[news.title.notnull()]

        #######
        ### Add news dataframe to PostgreSQL
        #######
        conn = establish_db_conn()
        cursor = conn.cursor()
```

```
cursor.execute(
    '''CREATE TABLE IF NOT EXISTS stockinfo.stocknews
    (
    symbol VARCHAR(4) NOT NULL,
    title TEXT,
    description TEXT,
    author TEXT,
    article_date DATE NOT NULL,
    url TEXT,
    PRIMARY KEY(symbol, title)
    );
    '''
)
conn.commit()

# drop the constraint if it already exists so adding doesn't cause an error
cursor.execute("""ALTER TABLE stockinfo.stocknews
                DROP CONSTRAINT IF EXISTS unique_symbol_title;""")
conn.commit()

# add constraint
cursor.execute("""ALTER TABLE stockinfo.stocknews
                ADD CONSTRAINT unique_symbol_title UNIQUE (symbol, title);""")
conn.commit()

for index, row in news.iterrows():
    query = '''INSERT INTO stockinfo.stocknews
        (
        symbol,
        title,
        description,
        author,
        article_date,
        url
        )
        VALUES(
        %s,
        %s,
        %s,
        %s,
        %s,
        %s
        )
        ON CONFLICT (symbol, title) DO NOTHING;'''
```

```
            args = (
                row["ticker"],
                row["title"],
                row["description"],
                row["author"],
                row["date"],
                row["url"],
            )
            cursor.execute(query, args)
            conn.commit()

        conn.close()

    def get_stock_names():

        # List of companies and their ticker symbols
        companies = pd.DataFrame([
            ('Pfizer', 'PFE'), ('Merck', 'MRK'), ('Johnson & Johnson', 'JNJ'),
            ('Amgen', 'AMGN'), ('Biogen', 'BIIB'), ('Gilead', 'GILD'),
            ('Medtronic', 'MDT'), ('Abbott Laboratories', 'ABT'), ('Becton, Dickinson and
Co', 'BDX'),
            ('UnitedHealth Group', 'UNH'), ('HCA Healthcare', 'HCA'), ('Humana', 'HUM')
            ],
            columns=["name","symbol"]
        )

        conn = establish_db_conn()
        cursor = conn.cursor()

        # Create the stock names table if it doesn't exist
        cursor.execute(
            '''CREATE TABLE IF NOT EXISTS stockinfo.stocknames
            (
            symbol TEXT PRIMARY KEY,
            name TEXT
            );
            '''
        )
        conn.commit()

        # Add symbol names dataframe to PostgreSQL
        for index, row in companies.iterrows():
            query = 'INSERT INTO stockinfo.stocknames (symbol,name) VALUES(%s,%s) ON CONFLICT
(symbol) DO NOTHING;'
```

```
            args = (row["symbol"],row["name"],)

            cursor.execute(query, args)
            conn.commit()

        conn.close()

    def get_date_table():

        # Define required parameters
        api_key = "apikey=5036b66d0f6fd3c5eb559e1c2abade9b"
        api_request_open_market = f"https://financialmodelingprep.com/api/v3/is-the-market-
open?{api_key}"

        # Get the response from the API and create a dataframe of holidays from the response
        response = requests.get(api_request_open_market)
        if response.status_code != 200:
            raise ValueError(f'Failed to fetch date table: {response.status_code},
{response.text}')
        df_holidays = pd.DataFrame(response.json()['stockMarketHolidays']).iloc[:,1:]

        # Initialize a closed days list
        closed_days = []

        # for each day the market is closed, add it to the closed days list
        for year in range(len(df_holidays.iloc[:,0])):
            for day in range(len(df_holidays.iloc[0,:])):
                if df_holidays.iloc[year,day] is not None:
                    #closed_days.append(df_holidays.iloc[year,day])
                    yyyy = int(df_holidays.iloc[year,day].split('-')[0])
                    mm = int(df_holidays.iloc[year,day].split('-')[1])
                    dd = int(df_holidays.iloc[year,day].split('-')[2])
                    closed_days.append(date(yyyy,mm,dd))

        # Get the start date and end date for filtering
        start_date = get_start_date()
        end_date = get_end_date()

        # In the date list, go through all the dates between
        # start and end date and add those to the date list
        date_list = []
        current_date = start_date
        while current_date <= end_date:
            if current_date in closed_days:
                temp = [current_date, "Closed"]
```

```
        elif current_date.weekday() >= 5:
            temp = [current_date, "Closed"]
        else:
            temp = [current_date, "Open"]
        date_list.append(temp)
        current_date += timedelta(days=1)

    # Make the datelist a dataframe of date and if the market was open or closed
    date_table = pd.DataFrame(date_list, columns=['Date','Market Status'])

    conn = establish_db_conn()
    cursor = conn.cursor()

    # Create a table of dates and add to postgresql
    cursor.execute(
        '''CREATE TABLE IF NOT EXISTS stockinfo.dates
        (
        market_date DATE PRIMARY KEY,
        market_status VARCHAR(6) NOT NULL
        );
        '''
    )
    conn.commit()

    for index, row in date_table.iterrows():
        query = '''INSERT INTO stockinfo.dates
            (
            market_date,
            market_status
            )
            VALUES(
            %s,
            %s
            )
            ON CONFLICT (market_date) DO NOTHING;
            '''
        args = (
            str(row["Date"]),
            row["Market Status"],
        )

        cursor.execute(query, args)
        conn.commit()

    conn.close()
```

```
    # Define operators
    get_date_table_task = PythonOperator(task_id='get_date_table',
python_callable=get_date_table)
    load_stock_recs_task = PythonOperator(task_id='load_stock_recs',
python_callable=load_stock_recs)
    get_stock_info_task = PythonOperator(task_id='load_stock_info',
python_callable=get_stock_info)
    get_stock_news_task = PythonOperator(task_id='load_stock_news',
python_callable=get_stock_news)
    get_stock_names_task = PythonOperator(task_id='get_stock_names',
python_callable=get_stock_names)
    check_db_health_task = PythonOperator(task_id='check_db_health',
python_callable=check_db_health)
    null_task = DummyOperator(task_id='dummytask')

    # Define flow
    check_db_health_task >> null_task
    null_task >> load_stock_recs_task
    null_task >> get_stock_info_task
    null_task >> get_stock_news_task
    null_task >> get_stock_names_task
    load_stock_recs_task >> get_date_table_task
    get_stock_info_task >> get_date_table_task
    get_stock_news_task >> get_date_table_task
    get_stock_names_task >> get_date_table_task
```

## Flask REST API Application

```
from flask import Flask, request, jsonify
import pandas as pd
import requests
#import sqlite3
import psycopg2
import json

app = Flask(__name__)

# Establish a connection to the sqlite3 database:
#conn = sqlite3.connect('airflow.db')
conn = psycopg2.connect(dbname='jhu')
cursor = conn.cursor()

# Define a GET endpoint to query data:
@app.route('/api/query', methods=['GET'])
def query():
    # Retrieve table name from params
    table = request.args.get('table')

    # Map columns:
    datemapping = {'stockinfo.ratings': 'rating_date', 'stockinfo.stockinfo': 'info_date',
'stockinfo.stocknews': 'article_date', 'stockinfo.dates': 'market_date'}

    # Fail if a table is not given:
    if table is None:
        return jsonify({'Error': 'table parameter is required. Add "table=[table name]" as a
api parameter'})

    # Get other potential arguments. If the allowed parameters aren't in the additional
arguments
    if len(request.args) > 1 and 'stock' not in request.args and 'startdate' not in
request.args and 'enddate' not in request.args:
        return jsonify({'Error': 'parameters must be "stock", "startdate", or "enddate"'})

    # Get the additional arguments
    stock = request.args.get('stock')
    start_date = request.args.get('startdate')
    end_date = request.args.get('enddate')

    # Create the query, adding where clauses as required
    query = f"SELECT * FROM {table}"
    if stock is not None or start_date is not None or end_date is not None:
```

```
        query = query + " WHERE "
        if stock is not None:
            query = query + f"symbol = {stock}"
            if start_date is not None or end_date is not None:
                query = query + " AND "
        if start_date is not None:
            print(table, ',')
            query = query + f"{datemapping[table]} >= {start_date}"
            if end_date is not None:
                query = query + " AND "
        if end_date is not None:
            query = query + f"{datemapping[table]} <= {end_date}"

    # Query the database into a dataframe and then return a json of the information
    df = pd.read_sql_query(query, conn)
    data = df.to_json(orient='split', index=False)
    parsed = json.loads(data)
    return json.dumps(parsed,indent=3)


if __name__ == '__main__':
    port = 8001
    app.run(debug=True, port=port)
```