StockTracker APP: Inner Workings

I often find myself scrolling through various subreddits after work to catch up on the latest articles about stocks. However, with so many subreddits to monitor, it can be challenging to identify which stocks are trending and deserve my attention.

Then I had an idea: as a programmer, I can automate this process. While I'm keeping the code for this project private, I'll explain the underlying mechanics and share snippets of the implementation in this documentation.

Tech Stack:

I chose Python for this project because it excels at automating daily tasks, especially for smaller applications. Although my primary expertise lies in backend .NET development, it's crucial to select the right framework for each use case.

Running the application manually every day would be cumbersome and could result in missed or faulty data. I needed a solution that could run autonomously at the same time each day. During my internship, I experimented with Azure Functions, and I realized it would be an ideal fit for this application.

Database Considerations:

Since I did my bachelor's thesis on cloud databases I'm well aware that databases can be expensive, primarily because most cloud databases are "managed," with costs heavily weighted toward that management aspect rather than just storing data. For this small project, performance wasn't a critical concern; I could afford for it to take up to 10 minutes to run without affecting the outcomes. Thus, my focus was on optimizing costs.

Previously, I've run databases like Neo4J in Azure Container Apps, but since my application doesn't require substantial resources, I opted for a simpler solution. I chose SQLite3, a lightweight database ran on the same client as the software.

You might wonder whether this choice means all data is lost every time the function executes, given that it operates on the same device as the software and functions delete temporarily stored files after completion. The answer is no. Performance wasn't a priority, so I manually created the database schemas using a maintenance Python script. I then uploaded these `.db` files to Azure Blob Storage. When the function triggers, it first retrieves the two `.db` files: one containing historical stock data and the other containing the list of words to be excluded.

Gathering Data:

The function is scheduled to trigger at 7 PM UTC, which corresponds to 8 PM for my local time, which is around the time I get home and would like my most up-to-date report. This schedule is defined using a CRON expression. To ensure I stay informed about any potential issues, I log additional information if the function experiences a delay in triggering.

```
@app.schedule(schedule="0 0 19 * * *", arg_name="myTimer", run_on_startup=False,
              use_monitor=False)
def StockTracker_Main(myTimer: func.TimerRequest) -> None:
    if myTimer.past_due:
        logging.info(f'Timer is past due. Executing CreateReport at {datetime.now()}.')

    CreateReport()
    logging.info('CreateReport schedule has finished.')
```

Upon execution, a `BlobServiceClient` is instantiated to retrieve the necessary database files. Once the database is set up, today's data is collected and thoroughly processed by the code.

```
blob_service_client = BlobServiceClient.from_connection_string(os.getenv('AzureWebJobsStorage'))
```

Once the database is set today's data is collected, all of this data will be thoroughly processed by this part of the code:

```
for submission in today_posts:
    title = submission.title
    body = submission.selftext
    submission_words = (title + " " + body).split()

    process_word_collection(submission_words, cursor_exclusion, post_mentions, submissions_stock_collection)

    submission.comments.replace_more(limit=0)
    for comment in submission.comments.list():
        body = comment.body
        comment_words = body.split()
        process_word_collection(comment_words, cursor_exclusion, comment_mentions, comments_stock_collection)
```

I retrieve all new posts from 7 PM yesterday to 7 PM today (according to Function Server time). The collected words are first cleaned of non-ASCII characters; for instance, variations like `$NVDA`, `"NVDA"`, and `NVDA!` will all be standardized to `NVDA`. Only uppercase words that are between 2 and 5 characters long are retained.

Next, each of these words is checked against the exclusions database, as many words fitting this criteria are not stock tickers (e.g., "YOLO", "LMAO"). If a post mentions a ticker multiple times (e.g., six times for a single ticker), it should only count as one post about that stock. To accomplish this, I use a set to collect unique words from each post.

Finally, this set is looped through, and the mentioned stocks are added to a collection specific to that subreddit, as well as to a total collection for the overall report. Both collections are essential for accurate data representation.

This process is run for every post followed by every comment on that post.

Blob Upload and Email Generation

If the function runs according to the schedule (as opposed to being triggered manually for development), the blob with the updated data is uploaded to the cloud.

All gathered data is utilized to generate HTML code and graphs using the Matplotlib library, culminating in a well-formatted email. Once the email is fully prepared, it is not sent immediately because I wanted the possibility for people to request a report on demand so I need to save the latest report in a separate blob.

When the function is triggered by the schedule, the email is sent using a disposable Gmail account to recipients whose addresses are hardcoded into the program. Otherwise the email is sent to the addresses that were in the JSON request.

The functionality for requesting reports will be implemented on my website, allowing anyone to request a report at their convenience.

Exclusions Management

Lastly, I have implemented an endpoint for updating the exclusions database. Once the application is populated with data, I cannot simply recreate a new exclusion database locally. This is because, when an exclusion needs to be added, I want every mention of that stock to be purged from the database. The endpoint for this function is set to private, ensuring that my database cannot be tampered with from external sources.

Challenges Encountered and Lessons Learned

Understanding Function Architecture:
I initially believed that a single Function App could contain multiple function files, compiling only the essential code. However, after creating my `SendMail` function, I discovered that my other function had been deleted after deployment. I learned that a Function App is essentially one file, and if I want multiple triggers within a single Function App, they must all be defined within the same `function_app.py` file. This means that when the email-sending function is executed, the code for generating the report is also compiled. While this approach may not be optimal for efficiency, it is more cost-effective than creating multiple Function Apps that call each other. Thus, consolidating all three endpoints into the same codebase emerged as the best solution.

RunOnStartup Variable:
I spent some time troubleshooting why the function was running twice. I had always assumed that the `RunOnStartup` variable, passed in the function trigger parameters, was intended for development purposes. In reality, it is used to trigger the function during its first cold start. A cold start occurs when the function has not been triggered for an extended period, causing it to trigger immediately upon startup and again when the scheduled condition is met. Since my function is triggered once a day, it always cold started, leading to it running when waking up and again immediately after, resulting in various issues.

Ensuring Comprehensive Data Capture:
One of the most significant logical errors I made was in the data I added to the database. I initially included only the stock tickers mentioned in posts and tracked their frequency in comments. However, I overlooked stock tickers mentioned solely in comments. To rectify this oversight, I implemented additional logic to ensure all relevant tickers were properly recorded in the database.

```
all_tickers = set(post_mentions.keys()).union(comment_mentions.keys())
```

Function Self-Termination

The most significant issue I encountered was that while everything was functioning perfectly in my local environment, taking about two minutes to execute, the Azure function frequently terminated unexpectedly in the cloud without providing a clear reason. The error logs showed minimal information, only indicating that “drain mode activated.”

```
Connected!
2024-11-02T12:41:57Z   [Error]   Executed 'Functions.StockTracker_Main' (Failed, Id=c0d081db-4d0c-4d01-85e8-fc2b268e547a, Duration=116692ms)
```

Initially, I suspected that drain mode was activating too soon and spent excessive time investigating this issue. However, when I shifted my focus from the log stream to the invocation history of the function, I discovered the root cause of the problem: a path-finding error related to image storage.

I had created a directory within the code to temporarily store images before adding them to the HTML. This approach worked seamlessly in my local environment because I was running the code from the correct directory. In contrast, the Azure function executes from a different directory, which led to the function being unable to locate the manually created directory. To resolve this, I needed to implement code that dynamically creates a `GraphImage` folder inside the `tmp` directory, which is typically used by Azure Functions for temporary data storage.

Gmail not support embedded HTML:

Everything seemed to be working perfectly until I opened the email on my phone (Gmail app), and suddenly, no images showed up. This was strange because, on my PC (Outlook app), everything worked just fine.

After some research, I discovered that Gmail does not support embedded HTML images in both the Gmail app and Gmail in a browser. Other email clients also have restrictions on embedded HTML images.

Solution:

embed the images like this in the for loop:

```
cid = f"image{index}"

html_content += f"<img src='cid:{cid}' alt='{stock} Mentions Graph' style='width:700px; height:auto;'></div>"
```

Then save the cid's in an array, pass it to the mail sending function and create the email like this:

```
msg.attach(MIMEText(html_content, "html"))

for file_path, cid in image_cids: with open(file_path, "rb") as img_file:
        img_data = img_file.read() img = MIMEImage(img_data, _subtype="png")
        img.add_header("Content-ID", f"<{cid}>") img.add_header("Content-Disposition",
                                                "inline",filename=os.path.basename(file_path))
        msg.attach(img)
```

This approach attaches the images without directly embedding them into HTML, which works across email clients. However, there was an issue: I wasn't passing the HTML to the `SendEmail` function. Instead, I uploaded the HTML with the embedded images into Blob Storage so that it could be retrieved later by the Email Function.

Alternative Solution:

The only alternative would be to upload each image (around 30) to Blob Storage as well, so they can be retrieved alongside the HTML. However, I decided against this approach because that's a lot of data stored and alot of requests. However I can't just add a note to my portfolio website saying "You can download the report here, just not for Gmail."

Instead, I created a fourth function that:

- Accepts a simple GET request.
- Downloads the HTML from Blob Storage.
- Converts it to a PDF.
- Sends the PDF back to the client.

This way, visitors to my site can download a PDF version of the report instead of requesting it by email.

I'm not letting the frontend download directly from Blob Storage because it would complicate security significantly.