Produced by Nathan Woolley

StockBuddy – a stock trading app that shows what other users are following. It also shows live stock market data.

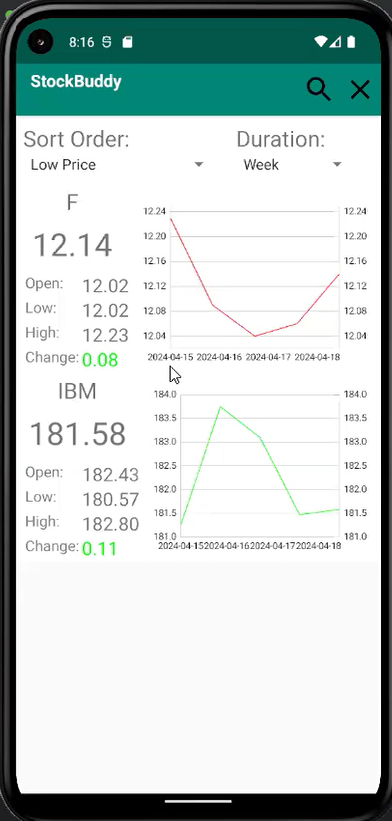
The demo video is found here:

<https://drive.google.com/file/d/16ahXHT8jFDK2W5ShSlI0EfWk_8oJage3/view?usp=sharing>

The github hosting this repository is found at two locations:

<https://github.com/ut-msco-s24/woonex-final-project>

<https://github.com/woonex/utap-final-project>

This app uses the common android apis that we used during the class. The substantial ones are recycler views, live data, and view models. The other apis that were used were from finnhub.io and alphavantage.co. Finnhub provides the data for the current stock data and the supplementary data such as earning. Alphavantage provides the live data for the charts. Finnhub provided a third party Kotlin library that was mostly helpful for collecting the data. However, it was difficult sometimes because their symbols and data for some of the most common calls were single letter abbreviations. The call providing the live data was especially bad – see Figure 1 - Finnhub Doc. They had the opportunity to change the names in Kotlin but didn’t do it and left the variables as the same name as the documentation data.

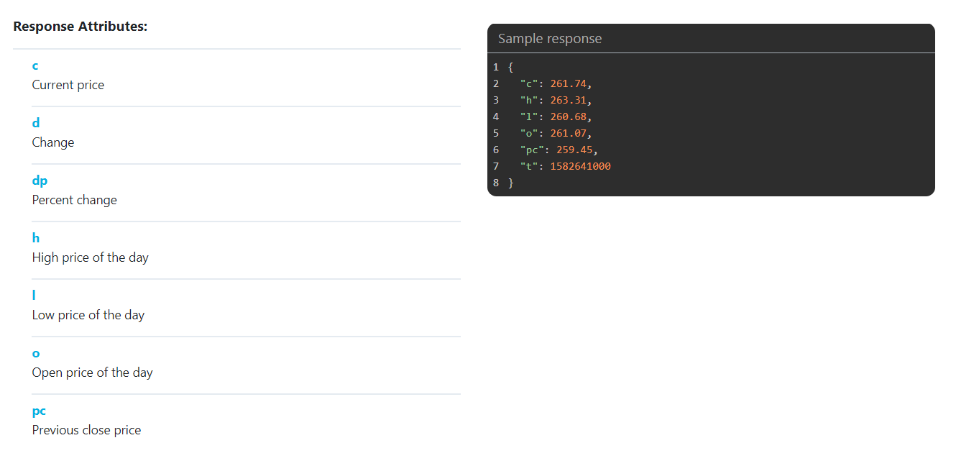


Figure 1 - Finnhub Doc

Another third party library I used was from <https://github.com/PhilJay/MPAndroidChart>. This a very well structured library that provided straightforward calls and methods to load and add the data. I used a forum post to format the data on the axis mentioned here: following the approach from here by Yasir-Ghunaim commented on Aug 22, 2016: <https://github.com/PhilJay/MPAndroidChart/issues/789>.

The final third party library that I was using was Firebase with Firestore for database. This manages the users favorite selection and shows other users the amount of followers following the current stock. I initially considered using separate documents for each user, but then that made querying more difficult. Using a query from the database with a single collection allowed me to filter either by user or by stock name.

I largely wrote most of the software by myself, but used a little help from AI for one logic section. The data displayed on screen required chaining across a number of different calls and across some separate APIs. This created a bottleneck for the screen lagging because it would have to fetch the name from the abbreviation, then the recommendations, then the similar stocks. ChatGPT suggested a fix found on line 185 in MainViewModel where it allows to build a favorite asynchronously. This was not something that I would have though of but it significantly reduces the time.

I was particularly proud of the UI/UX design of the top bar that hides or shows only the relevant actions to the user (hiding the favorites navigation if we’re already there, hiding the search if we’re already searching)

There were a couple of noteworthy backend components that I was proud of:

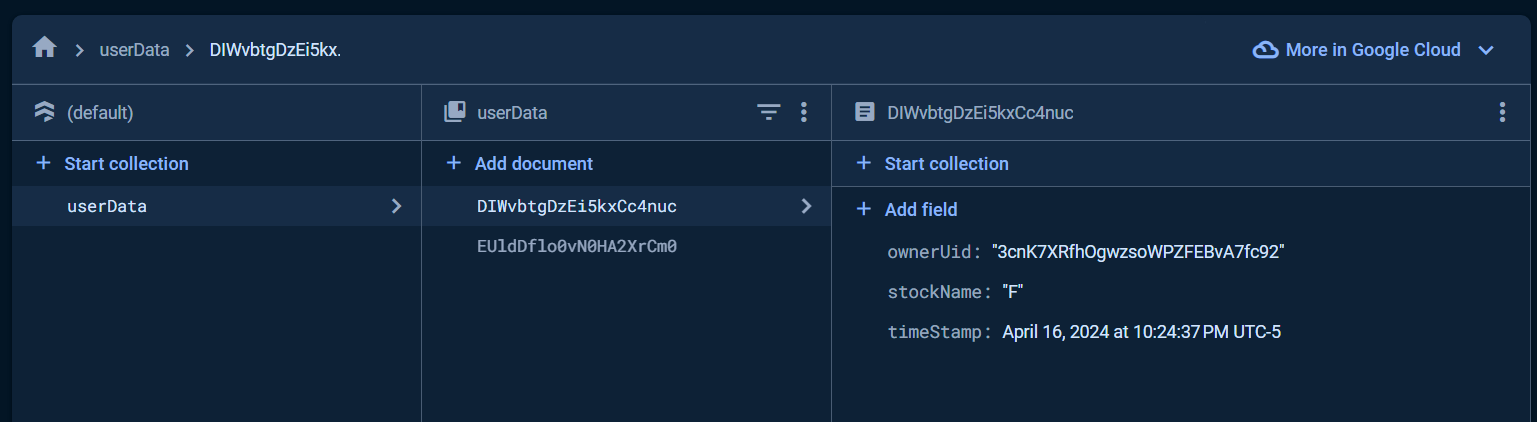
1. The apis that I was using were publicly available, but had a rate limit. Instead of fetching all the data each time that I was loading a screen, I maintained more persistence for data that is more persistent. For example, the earnings are published only once every 3 months so we know that we can cache that locally. The Expert Recommendations are also only updated every month or so along with the name. However, I still fetch the live price each time that the stock is called to live query the data since it’s an important piece of data that can rapidly change.
   1. One portion where I had to compromise on this was the API provided by Alphavantage. Their rate limit is exceedingly low (25 per day). This required capturing the data from all the historical data available and storing it memory. However, 25 per day is a very low amount and so it might switch to the example data if I’ve exceeded that. I reached out twice to contact them about an educational license that would have more data available, but never received a reply.
2. As mentioned earlier, I use Firestore for a database of managing users. I initially thought that Firestore restricted query of the data as limited to the current user only, but this was an incorrect assumption. Firestore allows you to query the entire database and it’s up to the developer to filter the data down correctly. This was critical to displaying favorites that the current user has selected while also displaying the quantity of other StockBuddy users following a stock.

One of the most interesting things about this project was managing and understanding which data needs to be stored where and what dependencies the data has. All stocks are largely keyed off in this app by their abbreviation. This means that all requests need to have that data loaded and any persistent data should be keyed off of the abbreviation.

One of the most challenging concepts of this app was the layout. It was a long and difficult setup to present the data in a useful manner and just setup all of the different items that are conveying information. There were some parts that were simplified by reusing fragments within other fragments. One example is res/layout/fragment\_name\_abbr\_price.xml file. This was helpful for displaying stock data for both the stock that the user is interested in and also the related stocks to each of those. It simplified creating a layout and helped to create more of a theme. This app does display all the information that’s necessary for a user to follow stocks, but I’m not a UI designer so it feels a little bit plain sometimes.

The most interesting debugging story was in dealing with the live data. Observing live data for the data meant that there was three times that the view was called and I wasn’t aware of the second time it was being called. The sequence was that it was first what I set it at, the second time when the initial dependent live data was called, and then the time when the upstream live data was loaded. The second in the sequence was rapidly called after the first initialization which took a while to debug and handle it appropriately as it was basically bypassing the initialization that I had already done.

Building and running the project requires just running the app. Setting up the backend services requires a standard Firebase and Firestore twin setup. Additionally, this project requires an API key from both Alphavandage and Finnhub. The keys are used in their respective API files found in the package   
“io.woonex.stockBuddy.api”. I’ve included my demo keys, but new ones can be generated for free. A picture of the Firestore database schema follows below:



Here’s a final list of the code lines reported by cloc, however, there was one file that largely from the FC8 for the authorization user found in io.woonex.stockBuddy.AuthUser. The total lines of code from that file are 6, 16, and 90.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Language | Files | Blank | Comment | Code |
| Kotlin | 22 | 249 | 105 | 1381 |
| XML | 29 | 201 | 31 | 1316 |
| SUM | 51 | 450 | 146 | 2697 |

Code frequency graph:

