Stock Prediction App

Implementation Design

CS361 HW 4

Open	52.28	Div yield	-
High	52.81	Prev close	51.71
Low	49.47	52-wk high	52.81
Mkt cap	56.07B	52-wk low	19.05
P/E ratio	266.77		

Team 16:

Felix Brucker
Robert Detjens
Remi Kendig
Dominykas Zobakas
Lyell Read

Table of Contents

Specifications

Functional Specifications

1. A User can search for a stock by its ticker symbol. (e.g. INTC)
2. A User can search for a stock by its name. (e.g. Intel)
3. A User can search for a stock by category. (e.g. Technology)
4. The Software will get the selected stock's historical values for the last 1 year.
5. The Software will analyze the historical data to generate a prediction.
6. The Software will display the prediction from the above analysis along with the historical data in a line graph.
7. A User can create an account.
8. A User can save a stock to their account.
9. A User can login to their account to go to their dashboard.
10. A User's saved stocks will show up on their account dashboard.
11. A User can click on a stock on their dashboard to go to that stock's prediction page.

Non-functional Specifications

1. The Software will use Alpha Vantage[1]'s API to get stock data.
2. The Software will use an LSTM algorithm[2] to predict stock value.
3. The Software will use Python / TensorFlow to generate the prediction.
4. The Software will use HTML / CSS / JS to provide a website interface.
5. A User will be able to login in under 5 seconds.
6. The Software will display the prediction to the User in under 5 seconds[3].

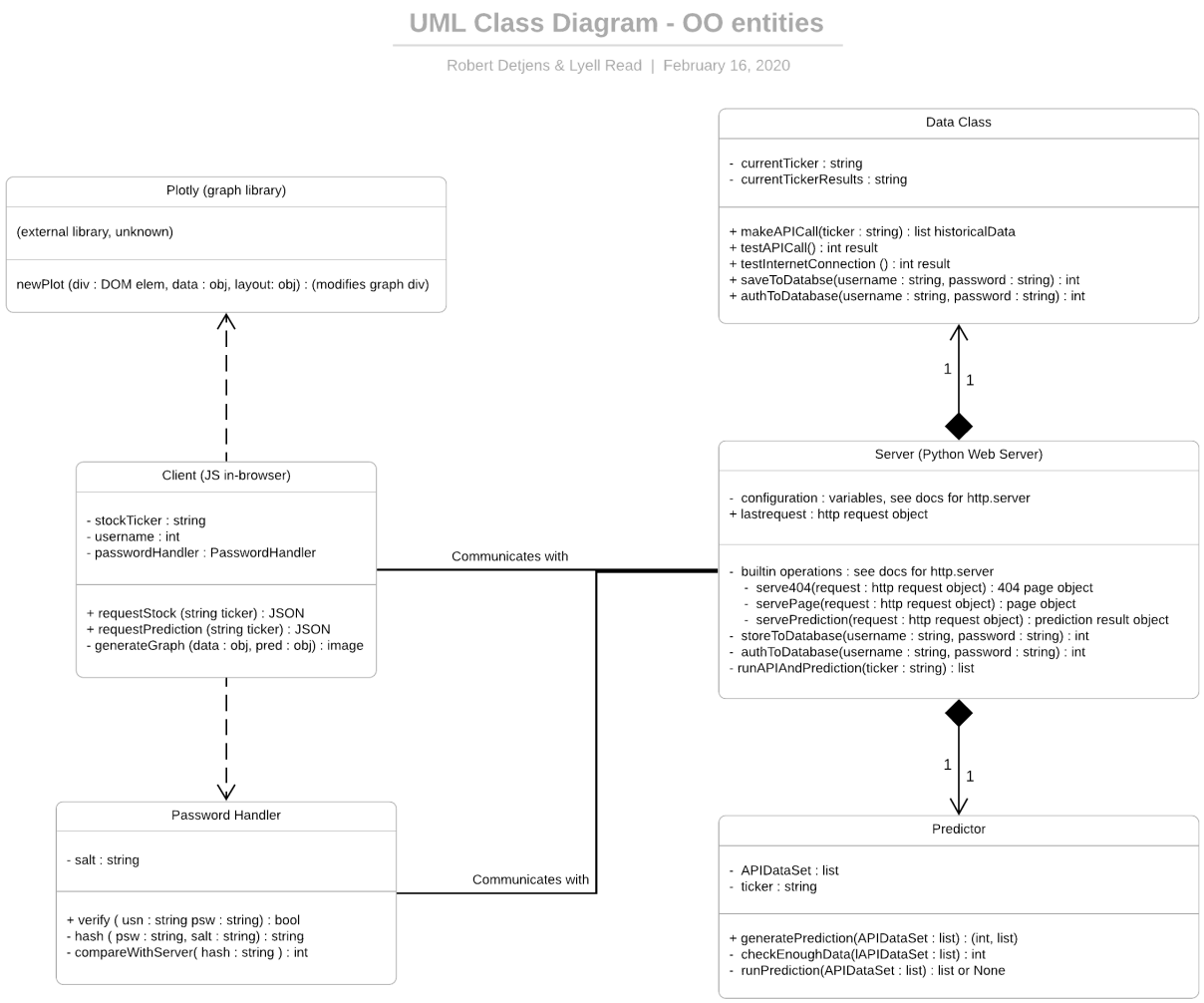
[1] https://www.alphavantage.co/

[2] https://towardsdatascience.com/using-lstms-for-stock-market-predictions-tensorflow-9e83999d4653

[3] The time to generate a prediction is currently unknown.

UML Class Diagram

Of note about this class diagram, we will be implementing both the client and the server with help from several libraries. These libraries provide functionality that would be difficult or insecure to provide without them. These include a python HTTP server, a graph rendering library, and a hashing library. Therefore, the component data types and functions have been left out of these as there would be thousands, even tens of thousands of entries if we were to include these.



Entity Packaging

Coupling

- Users can modify their account data when they change their password or save a stock to their account.
- The server sends the data and predictions to the client to pass along to Plotly.
- The client passes along user input to the password handler to send to the server for verification.

Cohesion

- The data works in tandem with the stock prediction API (indirectly through the server) to create predictions.
- The data connects to the server in order to get to the stock API.
- The server sends the data to the stock API.
- The server and the client work together to provide the functionality for the website.
- The client works together with Plotly to produce and display the prediction graph.
- The password handler works with the server to allow users to log in.
- The server and client work to make the predictions work and be visible.

Incremental Development

The Stock Prediction software would best support incremental development. Incremental development means that different parts of the system will be built in a specific order. These parts rely on the previous part working in order to function. Incremental development is in opposition to iterative development, meaning that the whole system must be built bit by bit at the same time because they all work together. Access to the stock prediction API must be completed before predictions can be made. The website must be able to get and display data from the databases and API before all the code can be effectively tested, as it would be very hard to see if everything is working without having some sort of interface. In this way, there is a clear order in which the system must be developed: get a functional client side website working, get access to the API, test how the API works with the website. It might be possible to develop in a different way, but this is the most logical order for this software project and its requirements. Not every part of the system needs every other part to work, so the design best supports an incremental development style. Though it would be possible to do iterative design, it would be less efficient for this software project specifically.

Design Patterns

Builder (Used)

- Plotly is a third party builder that creates graphs in a separate action. This allows the application to quickly render visualizations of stock data for the user.

Adapter

- Not used. This is not an existing system we are modifying

Facade (Used)

- The server serves as a facade toward the client, obscuring the data class and predictor from it. It processes stock data and prediction data, but acts as the intermediary, with the client having no direct contact with the two classes behind the server.

Memento

- Not used. The state of the overall system is not saved, only portions of data within the system are saved e.g. saving stocks to a user's profile. Therefore, memento is not applicable.

Interpreter (Used)

- The client serves as an interpreter for the user. When the user interacts with the website the first and only point of contact is the client.

Observer (Used)

- The server serves as an observer of the client and of the password handler. It updates stock data so calculations are up to date, and verifies accounts. In the other direction, the client also observes the server, updating the user interface with new predictions and showing login.

Template

- Not used. Our single algorithm does not need to be handled incrementally.

Factory

- Not used. We do not need to build a variety of objects, at least within the scope of the given classes.

Strategy

- Not used. It would be possible to implement Strategy as a way of letting the user choose what algorithm processes the stocks, but it is not relevant with our use of only one algorithm.

Decorator

- Not used. No additional responsibilities after the fact are added to classes.

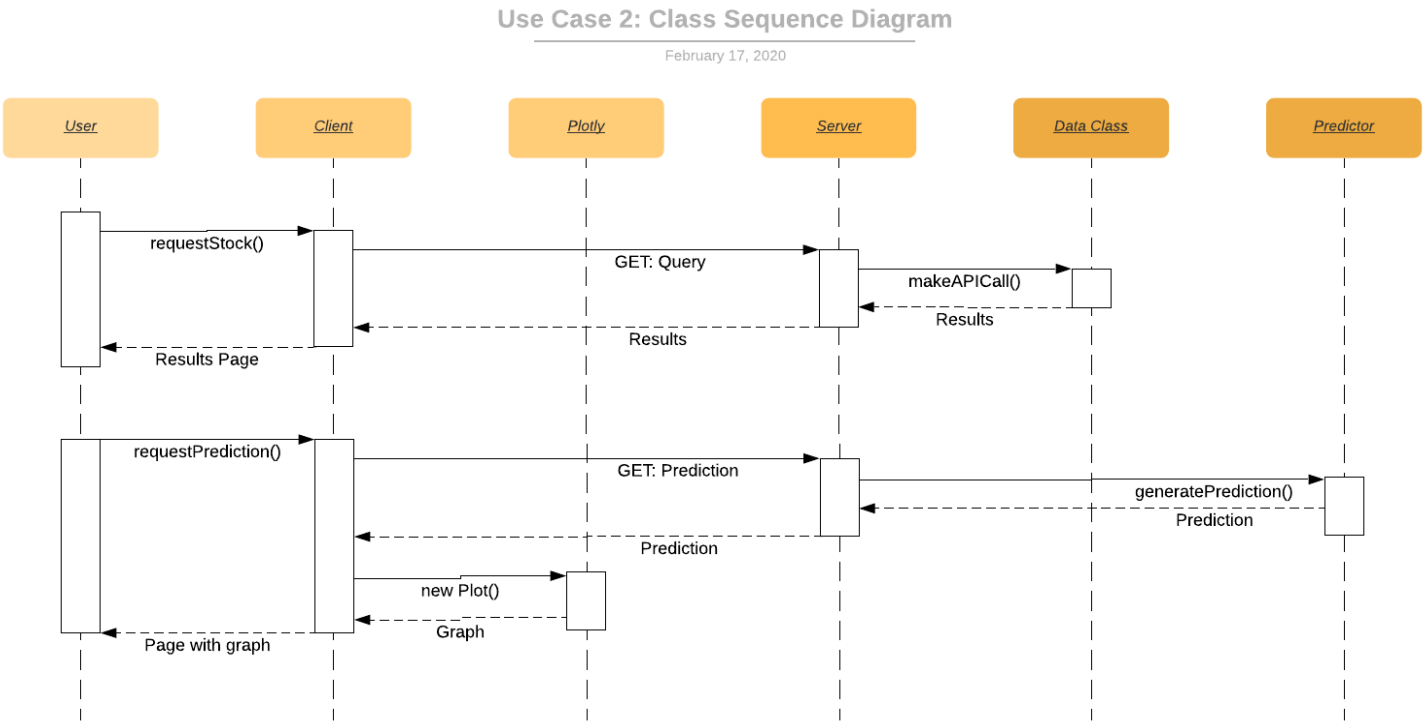
Composite

- Not used. There are no large sets of objects on this level of specificity that could be organized into composite. One could make the argument that various webpage files are composite to the client, however, but they are not of special significance to the class organization as that is standard practice for a webpage.

Visitor

- Not used. We are not modifying an existing system, and the architecture does not require a hovering visitor over existing classes.

Class Sequence Diagram for UC2



Interfaces

Server Interfaces

builtin operations

- Subroutines serve404, servePage, servePrediction handle backend client web requests.

storeToDatabase

- Stores login credentials from client to account database.

authToDatabase

- Returns authentication attempt result value back to the client, given credentials.

runAPIAndPrediction

- Requests data handling and prediction from Data Class and Predictor.

Data Class Interfaces

makeAPICall

- Request stock data from the exterior API.

testAPICall

- Do a test request for stock data before attempting genuine call. Essentially to verify connection to the API.

testInternetConnection

- Check for internet access before attempting other actions. Runs as part of *testAPICall* to test internet connectivity.

saveToDatabase

- Stores credentials from the request that came from client via server to the database.

authToDatabase

- Checks stored credentials and returns a value if they match, otherwise returns false (0).

Predictor Interfaces

generatePrediction

- Gives back a prediction given the provided batch of stock data.

checkEnoughData

- Confirms that the current batch of stock data is parsable and sufficiently large to make an accurate prediction.

runPrediction

- Is called to produce the prediction itself using the given algorithm.

Client Interfaces

requestStock

- Sends request to server for historical data for a given stock.

requestPrediction

- Sends request to server for a prediction for a given stock.

generateGraph

- Calls Plotly to generate a graph of the received data and prediction.

Plotly Interfaces

newPlot

- Generates a graph from given stock and prediction data.

Password Handler Interfaces

Verify

- Call other methods to safely confirm given credentials are correct.

Hash

- Hash the password for security when it is being passed through the system.

compareWithServer

- Compare given credentials to the server database so they can be authorized.

Exceptions

Given that the languages we will be using are intended to use exceptions and exception handlers as control flow managing structures[4], this will be reflected in the list of exceptions presented below.

InvalidQuery

- Caused by: User entering a query that is not exactly matched against the whitelist of available ticker symbols.
- Exception handler does: The exception handler fast tracks the return of a "search not found" message to the user. Once this message is sent, control flow resumes.

NoHistoricData

- Caused by: The API call that collects historic data for the given stock cannot find enough meaningful data on the given ticker to get a meaningful result from the prediction
- Exception handler does: The exception handler informs the user of this, and control flow returns to status quo.

PredictionAlgorithmFailure

- Caused by: Any error returned by the prediction algorithm. This can be anything from unable to find a proper prediction, to a syntax error in the prediction algorithm.
- Exception handler does: Handles the fact that there will be no prediction, informs the user, and returns control flow to usual. Might also notify the developers of the failure with diagnostic information.

HTMLNotFound

- Caused by: The backend server searches for a static HTML (or handlebars, depending on what route we take) and cannot find this file.
- Exception handler does: Attempt to find 404.html, and return that. Otherwise, the server is missing all the HTML files, so send a hardcoded 404 page (HTML is present in source code for server).

LoginFailure

- Caused by: Login failed for any of the following (password wrong, username wrong, either field empty, any integrity checks fail).
- Exception handler does: Return "not authorized" or "login failed" messages.

DatabaseNotFound

- Caused by: When the backend attempts to connect to the database to perform login operations, and cannot connect or find this database.
- Exception handler does: Notifies developers of the misconfiguration, and replies to all login attempts with Internal Server Error.

[4] https://en.wikipedia.org/wiki/Exception_handling

Contributions

Customer

HW4

- We did not meet with our customer as we had already verified that we share a common understanding of the goals and requirements for this app so far.

Team members

HW4

- Felix Brucker: Design Patterns, Interfaces
- Robert Detjens: UML Class Diagram, Interface Revision
- Remi Kendig: Incremental Development
- Dominykas Zobakas: Class Sequence Diagram
- Lyell Read: Document Formatting, Exception Handling, UML Class Diagram

Appendix A: Historical Contribution Log

Customer

HW1

- We met with our customer Ghaith Shan after class on 1/23/2020.
- We have maintained communication with our customer on Discord since 1/16/2020.

HW2

- We met with our customer Ghaith Shan after class on 1/30/2020 to present the paper mockups.
- We discussed requirement revisions and went over the mockups with our customer over Discord on 1/31/2020.

HW3

- We did not meet with our customer as we had already verified that we share a common understanding of the goals and requirements for this app so far.

Team members

HW1

- Felix Brucker: Document Creation, Data Flow Diagram, Proofreading
- Robert Detjens: Functional Specifications, Non-functional Specifications, ERD Revision, Use Case 2, Use Case 2 MSD
- Remi Kendig: Use Case 1, Data Flow Diagram
- Dominykas Zobakas: ERD Revision, Use Case 1 MSD
- Lyell Read: ERD First Draft, Document Formatting, Customer Contributions, Functional Definitions, Non-Functional Definitions, Use Case 3, Use Case 3 MSD

HW2

- Felix Brucker: Mockup proofing, Use Case 1 Revision
- Robert Detjens: Paper mockups, Mockup meeting w/ customer, Definition revision
- Remi Kendig: Changes made since HW1
- Dominykas Zobakas: MSD Revisions, ERD revision
- Lyell Read: Mockup proofing, Digital Mockup Adaption, Appendix, Corrections to UC1 & UC3 MSDs.

HW3

- Felix Brucker: Implications
- Robert Detjens: Architecture 2 Data Flow Diagram, Fault Trees, Document Formatting
- Remi Kendig: Quality Attribute Assessment, Architecture Validation
- Dominykas Zobakas: Decomposition
- Lyell Read: Architecture 1 Data Flow Diagram, Team Member Contributions, Interfaces Revision