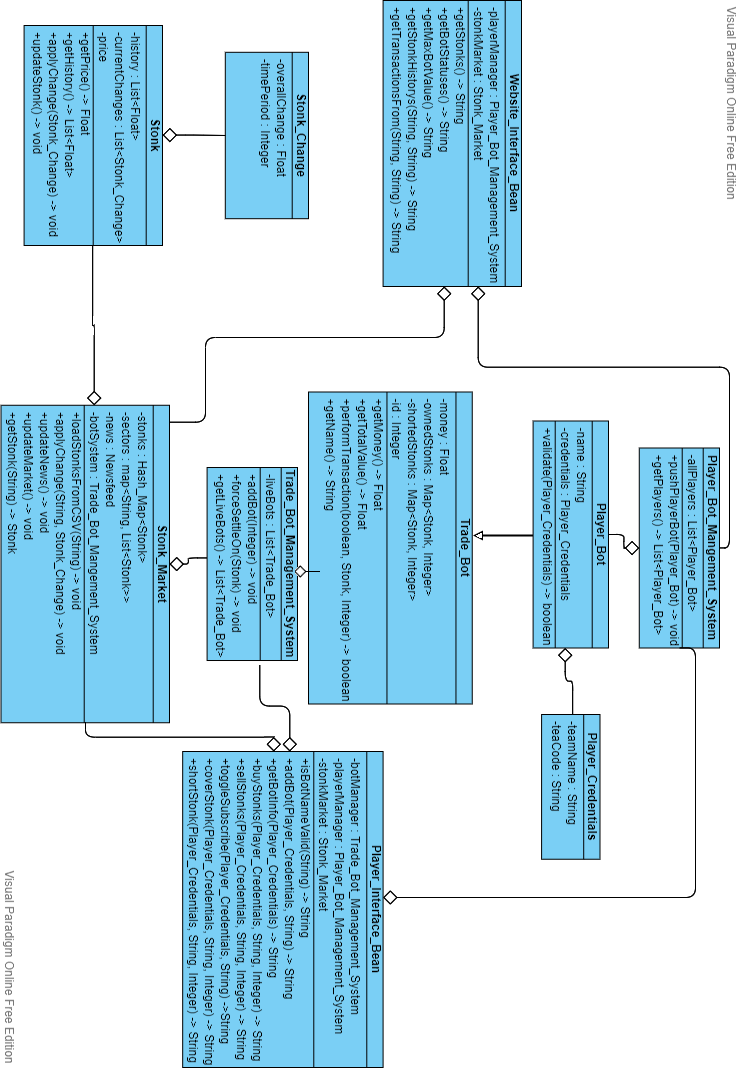
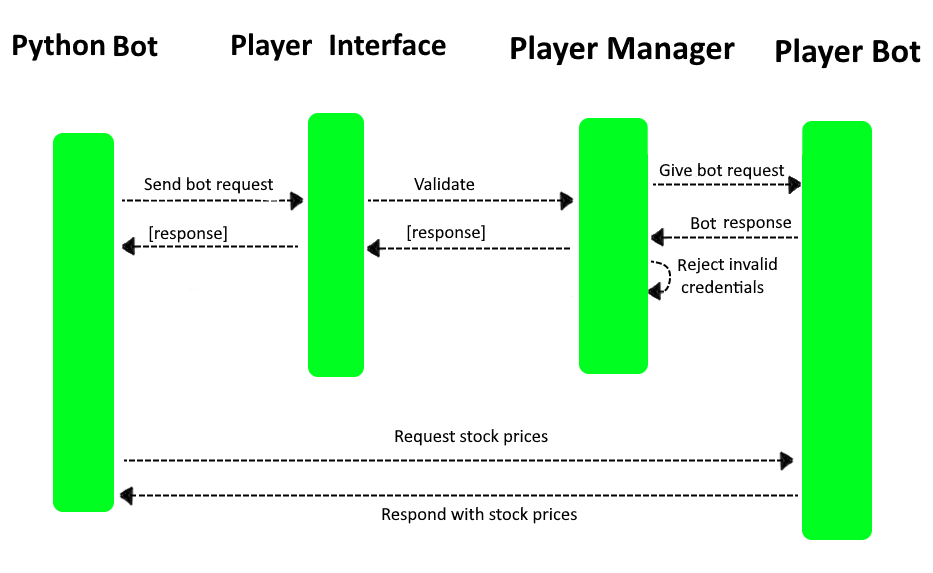
****

OO Design & UML

There are three main systems in our project. The Website and Player Interface Beans that handle http requests from our website and any player trade bots, the Stock Market (styled as stonk market to fit the colloquial humorous tone of the project) which runs a simulation of a virtual stock market, and the Bot System which updates both automatic trade bots (to help simulate a busy stock market) and polls bots that respond to player requests to update their information.

Trade bots contain information about their status in the stock market, including their current on-hand cash, stocks they own, and stocks they have shorted. They are stored in a bot manager that the stonk market can use to update all of them more easily. The stonk market also contains stonks which it creates from a CSV file, giving them their names, worth, and their *sector*, which allows an internal Newsfeed class (not shown in the diagram) to update entire groups of related stocks, simulating how entire industries may be identically affected by events in real life.

The stocks keep hold of these changes to them by means of the Stonk\_Change class, which tracks an overall change to the stocks value over a period. This allows the price of a stock to gradually increase or decrease in response to an event (news related, random event, trader interest, etc) rather than snap to a value, giving players time to react to the change. It also holds a list of past values in case players want to perform an analysis of the stocks value over time to try predicting future changes.

Player Bots are derived from Trade Bots and have only a few differences. The first is that they hold credentials, in the form of the team’s name of the team who wrote the bot, and a specific code needed to access the bot. *Any* HTTP request for that bot *must* include valid credentials of the bot in order to give it any commands. *This does not create any logs, nor cookies, as it is not remembered.*

The above is a dynamic diagram of how player written bots accessing the stock market shall do so. Note again that all requests to a bot must be validated to prevent rival teams from performing sabotaging acts. Also note that requests not changing the state of the bot, such as those forwarded to the stock market or about net worth do not require credentials as this is public knowledge.

**The context** in which we designed this system for was for a Computer Science Society Bristol event. The idea is that students who are capable programmers and/or are interested in the stock market and trading algorithms will meet up and form teams. They will then be given an hour to observe a virtual stock market while creating their own program that they shall run on their computers which will interact with the server the virtual stock market is on in order to control their own trader (Player\_Bot). After he full length of the event, the stock market shall close, and whichever teams have the highest net value at the end will win the competition.

**The reason** for designing the system in this way, with external bots communicating with our server was to try reducing complexity. If we were to run the bots in a scripting language on he same server that the stock market was running on, we would have to find a way to run this language within the java environment, as well as figure out how to submit this script from the website to the server. This would also mean we have to create a log in system, where we would then have to worry about cookie law. Therefore, while having to create a spring bean specifically for interactions between the stock market and the server seems like an unnecessary layer of convolution, it was actually the best we could do given our abilities.

To try help this split, we also designed Player\_Bot with a websocket feature so that players could subscribe to specific stock and be notified when there are any changes to it.

Furthermore, the separation of stock, the market, and the bots, was so that we could alter logic in these objects separately without having to change the overall system. For example, altering how a stock updates doesn’t have to effect the market, and trade bots keep a layer of abstraction from dealing with the stocks directly. The reason for keeping the Newsfeed object directly in the Stonk\_Market object was because the news system is very much an extension of the random change system, we also have in the Stonk\_Market. Therefore, it will only ever be used in the Stonk\_Market in private functions and will not have any external interactions. For this reason, we kept the newsfeed as an internal, masked class.

**We learnt** while doing this that it is important to design a system that is resistant to future changes in function. As the relation between various classes changes, and more functionality was given to certain classes, we realised that we could have split the classes up further. While a lot of detail is abstracted away in the static UML, the Stonk\_Market class has become very large, and deals with changes to sectors, stonks, news, and websockets with player bots. This could easily have been multiple classes.

We also realised that it is important to have a firm idea of what you are creating before focusing on the specifics. Due to the classes interactions, there are a few cases where multiple classes can do the same thing. Both Website\_Interface\_Bean and Player\_Interface\_Bean have methods to return information about the stock market, and within the Stonk\_Market class there are various methods that are all used in the end for the same thing. This is due to changes in the model and miscommunications between members of our group.