Version 1.0 - 11/02/2023

Architecture and Design Documentation

Real-time In-browser Stock Explorer (RISE)

Taiga Chang, Michael Chao, Isaiah Daye,
Skylar Homan, Anoop Krishnadas

1. Original Architecture Based on Requirements Specification

1.1 Summary of Original

Our architecture layout was originally a layered architecture. We had a simple design that involved having an API at the lowest layer. The Data Manager was the data collection system and contained the more complicated functions that were used to pull specific data and interpret it, as well as send the data to a Database. The Database Layer would only be accessed by the UI and Data Managers. The UI Manager layer would just call functions from the backend for the purpose of presenting the data to the user. The UI Manager would send information to the Display layer, which allows the user to interact with the system in an easy way to understand. Communication would have to pass through each layer (moving around the Database to request an update to the Database)

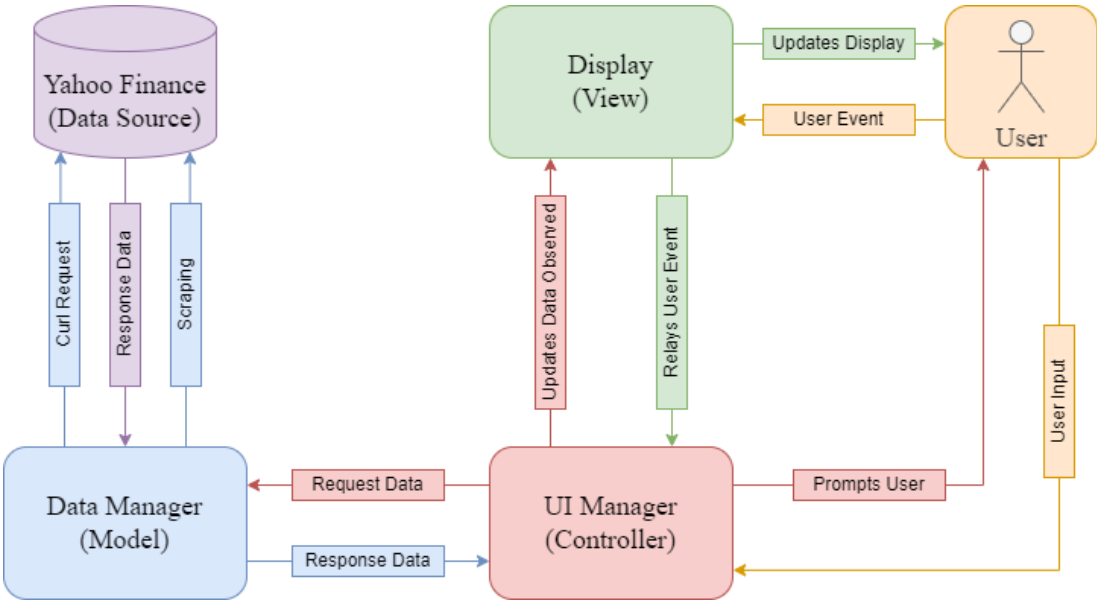
1.2 Outline of Changes

The biggest change in our data structure is our decision to move from layered architecture to a Model-View-Controller scheme. This was done because we realized that we had a more complicated framework than what we could achieve with layered architecture. Too many layers and wasted steps could slow down our website and prevent it from being real-time; for an application where real-time data display is the most important goal, this is not a great compromise. Instead, our new architecture has a Controller and Model adjacent to each other, with each directly contacting the other. The Controller can request data from the Model, send data to the View, and prompt the user via View elements. This MVC model is better suited for our project to increase efficiency.

2. Updated Architecture Patterns

2.1 Outline

Our application's architecture is based on a Model-View-Controller architecture pattern. The Display (View) passes user input to the UI Manager (Controller), which issues data requests and sends them to the Data Manager (Model). The Data Manager then retrieves data from an external source, interprets it, and passes it back to the UI Manager, which uses it to update the Display. Below is a simple diagram of our architecture; more detailed displays will be provided to show the smaller interactions between parts.



2.2 Meeting Requirement Specification

In terms of our requirements specification, we are removing the need for a database as it would overcomplicate the work we need to do. This is also due to the realization that it is unnecessary to add a database, since a traditional web application with pulls does not also require

a database feeding information to it. Other than the removal of the database, all other requirements remain the same. We have switched from an API to Beautiful Soup, a web scraper, for data retrieval, as this will still allow us to have the same final result and be able to reach the goal in time. Our functions list is attached below as Appendix A, but is outlined in more detail in our README File. Each of our interactions is built around the associated functions, but each function is not outlined in detail in diagrams because that requires exorbitant information.

2.3 Meeting Schedule/Deadline Requirements

Complications with accessing data from an API had originally set back the time schedule, but luckily we had the option of switching to Beautiful Soup instead of an API. This change is expected to have a significant schedule impact because of efficiency and preexisting familiarity with the library. Therefore, making these changes are not detrimental to our final product's result. With these changes, we were able to catch up to our intended schedule. To meet the deadline for the project, we will need to stay on track with our schedule and efficiently deal with any complications that arise. Our diagrams reflect a modification over a more ambitious project that we had specified in the original requirements specification. As a result, our project had consolidated the many parts originally outlined, allowing each object to have more functionality. With the changes made to our process, we may end up with the ability to add more features, which will lead to an update on this document.

2.4 Data Manager (Model) Interactions

The Data Manager, which is the Model component of the Model-View-Controller architecture, uses the web scraper Beautiful Soup to collect data from Yahoo Finance. The Data Manager receives data requests from the UI Manager (2.5), calls Beautiful Soup to obtain the necessary data, applies any necessary processing, and returns it to the UI Manager for display.

The Data Manager never directly interacts with the Display (2.6), allowing it to be optimized for efficient data retrieval and processing without worrying about display.

At startup, the Data Manager calls F6.2.2, start_clock(), to begin the application's internal clock. (See Appendix A for all function references.) At regular intervals, it queries this clock with F6.2.3, get_clock_state(), and executes F1.3.2, get_data(), if enough time has passed. get_data() is where the Beautiful Soup calls are made in order to scrape data on any focused and/or pinned stocks; since the Data Manager does not communicate with the Display, it obtains this list of stocks to retrieve information about from the UI Manager. A regular get_data() call returns the current share price of the stock, data points for the historical price graph, market cap, and other information (as detailed in the Requirements Specification). When this bulk data request is complete, the Data Manager notifies the UI Manager that the data has been updated.

Whenever the user provides input via elements of the Display, this information is passed to the UI Manager, which issues a request to the Data Manager for specific data. These requests are functions F4.4.3-8, the "get" functions. The Data Manager processes these by making a call to get_data() which only asks for the data specified by the "get" function(s) it was passed, then returns the output of these calls to the UI Manager for handling.

2.5 UI Manager (Controller) Interactions

The UI Manager, which functions as the Controller component of the MVC architecture, acts as an intermediate point of connection between the Data Manager (2.4) and the Display (2.6). The UI Manager receives all requests sent by the display elements, including form inputs such as search bar entries and button clicks, and determines what data is needed to fulfill those requests. It then sends its own request for that data to the Data Manager, which acts as necessary to retrieve or obtain that information. Once the Data Manager has found the data, it returns it to

the UI Manager, which uses the Next.js framework to inject that data into the appropriate elements of the Display, e.g. updating the current share prices displayed on pinned stocks in the sidebar.

At startup, the UI Manager calls F6.1.6, initialize_website(), and passes this function call to the Data Manager so it generates the initial site data, which can then be passed to the Display to render the initial page. The UI Manager is responsible for initiating function calls of F4.4.3-8, the "get" functions, in response to user input from page form elements passed by the Display. These calls are again passed to the Data Manager, which returns the expected data as detailed in 2.4. With this data, the UI Manager can optionally call F4.5.1, make_stock_graph(), and/or F4.5.2, make_stock_delta(), if either of those elements need to be created or updated. This intermediate processing functionality is kept out of the Data Manager to avoid holding up its other processes, since its primary responsibility is to fulfill data requests as quickly as possible. Finally, to complete the user response chain, the UI Manager calls F4.6.1-6, the "display" functions, which take whatever data was acquired and generated from the "get" and "make" calls and update the page contents to reflect the new data. At set intervals, the Data Manager will also automatically update its own data, passing this data to the UI Manager when it is ready. The UI Manager checks this data against the data currently in the Display and calls any necessary "make" and "display" functions. In addition, when the Display notifies the UI Manager of search bar input, the Manager calls F5.2.2, find_associated_tickers(), which again passes a request to the Data Manager for an array of stocks matching the search text and updates the page with the return data.

2.6 Display (View) Interactions

The Display, functioning as the View component of our architecture, is the HTML and CSS which the user's browser uses to render the application page, as well as the attached Javascript which manages user inputs. A variety of page elements are sensitive to user input: notably, the search bar, which takes text input; the sidebar buttons, which can pin, remove, or switch the focused stock; and the graph display window buttons, which change the interval represented by a stock's price graph. Each of these inputs are detected by the Display Javascript, and are then passed to the Controller component, the UI Manager (2.5). The Display itself does not change its own elements, because it does not directly receive data from the Model (2.4); rather, it is altered by the UI Manager and is responsible only for defining the appearance of the page.

As its embedded Javascript is relatively simple, the Display does not directly call any of the functions from the function list. Instead, it detects user input via the form elements and simply notifies the UI Manager of these inputs. All data processing and page updating is then handled by the UI Manager; the Display is no longer involved. This delegation of responsibility ensures that any potentially complicated or lengthy processing is done on the hardware hosting the application rather than in the user's browser, ensuring the application runs more quickly and does not slow down other web pages the user may be trying to use.

3. Structural Design Patterns

3.1 Data Manager (Facade)

Facade:

- Encompasses a large set of information with high complexity and simplifies it

- Acts as a command center for the information stored (objects, data, APIs)
- Client uses are limited to the Facade only

The client of the Data Manager is the UI Manager, which will interact with the Facade (Data Manager). The Data Manager interprets the interaction, makes decisions, and returns information based on those decisions. The Data Manager also acts as the command center, containing all the data collected from Beautiful Soup scraping, and parses/manages it for the UI Manager to use.

3.2 UI Manager (Decorator)

Decorator:

- Wraps application behaviors in processes which extend those behaviors
- Dynamically modifies features of another object

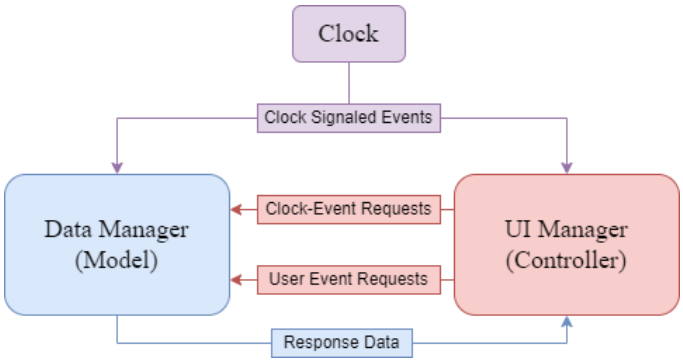
Structurally, the UI Manager uses the Decorator design pattern, as it wraps various application behaviors in processes which extend those behaviors. User events are recognized by the Display and relayed in order to request data on the selected stock from the Data Manager. However, the Data Manager only returns raw data; it is not formatted the way it needs to be to write it to the page, and it does not include the historical price graph. Before it returns the result of the initial request by writing data to the display, the UI Manager alters the output by reformatting the data and generating the graph. In this way, the UI Manager "decorates" each data request with a wrapper that fills in any functionality which is not explicitly called for, but is still required.

3.3 Clock (Bridge)

Bridge:

- Linkage between two orthogonal dimensions in the software
- Often links the user's needs to the platform services

Our Clock object matches a Bridge Design structure, as it acts as a coupler between the UI Manager and the Data Manager. The diagram below demonstrates the relationship between the clock and these two managers, where the clock makes underlying timed requests to control the interactions between the two.



4. Behavioral Design Patterns

4.1 UI Manager (Mediator)

Mediator:

- Mediator controls how a set of objects interact with each other
- The other objects pass functionality through the mediator

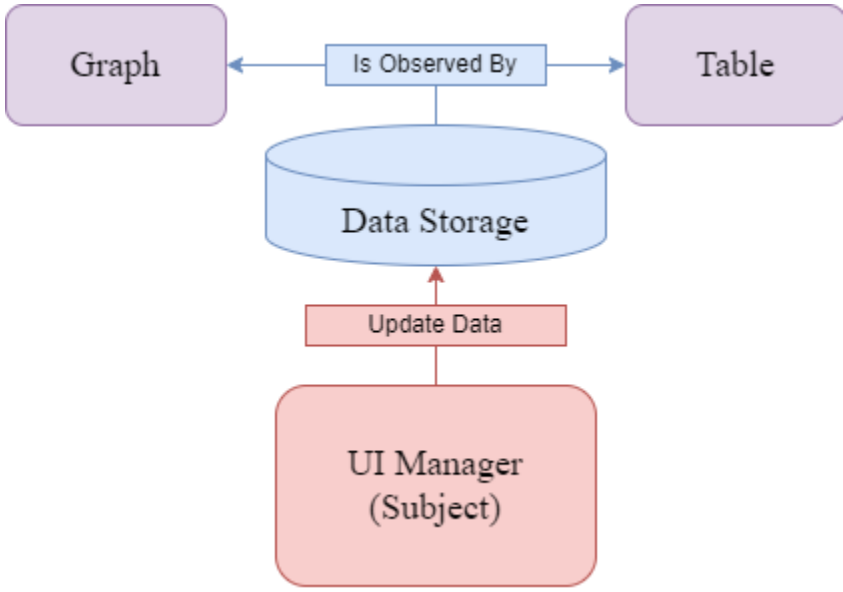
Behaviorally, the UI Manager utilizes the Mediator design pattern, as it functions as a layer of abstracted coupling between the back end (data processing/requests) and the user interface (website HTML and Javascript). It receives form inputs from the user interface and passes a request for the necessary data to the back end. When that request is fulfilled, it receives the data and handles the process of injecting it into the page contents. This allows back end data to be stored and handled in whatever format is necessary for processing, e.g. JSON objects or Pandas tables, creating independence in each part.

4.2 Graph/Chart (Observer)

Observer:

- Subject is the one manipulating the display by viewing the Observer
- Observer interprets multiple views (of data) and generates a model

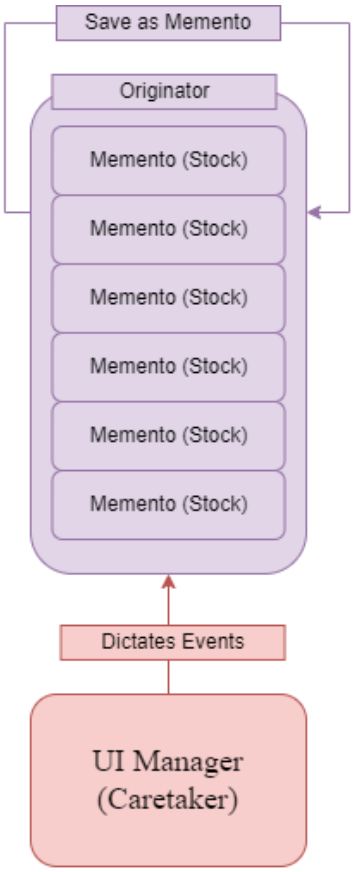
Whenever a stock is focused, the UI Manager, acting as the subject, compiles data into a graph and writes it to the display. This is not the only time the graph is updated: at set intervals determined by the application's internal clock, the Data Manager updates its data and notifies the UI Manager, which regenerates the graph and writes it to the display again. The graph is an observer of data provided by the Data Manager, and the model is manipulated by the subject (UI Manager).



4.3 Table (Observer)

Much like the graph (4.2), the table of additional stock data functions using the Observer pattern. As before, the UI Manager handles a subscription which updates the table every time the Data Manager changes the underlying data. This subscription and alteration is managed on a per-cell basis, so the entire table is not updated each time the Data Manager updates anything; cells are individually updated only when new data for them has actually been retrieved. In this way, the subject (UI Manager) manages the table's subscription to the underlying data and manipulates its model.

4.4 Sidebar Stocks (Memento)

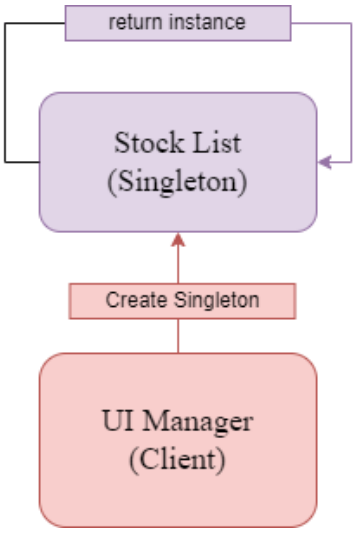


Memento:

- Requires the existence of an originator, a caretaker, and a memento
- Originator: The object that is able to save a memento
- Caretaker: The object that knows when the Originator needs to save/restore itself
- Memento: The object that stores the memento which can be written and read

The Sidebar Stocks object matches this definition, as the sidebar itself (Originator) is able to create memento objects of the different stocks that should be added to the sidebar. The Caretaker will be the UI Manager, basing its responsibility on user input or the clock, and the Memento will be an object that preserves the state of the stock.

5. Creational Design Patterns



5.1 Ticker List (Singleton)

Singleton:

- Singleton object will be initialized on first use, and will be the only instance of that item
- Singleton must also return itself or a pointer to itself when an instance is requested.

Our Ticker List only has one instance when the User initializes the website, allowing for preservation of data integrity. The Ticker List object resides statically within the UI Manager, and clients cannot access the singleton directly for modification, but by using accessor methods to search for stocks. Every time an instance of the Ticker List is requested, the same Ticker List object is returned/pointed to.

5.2 Website Display (Prototype)

Prototype:

- Prototype must have a defaulted format of the object and be able to be cloned
- Each instantiation of the class modifies a clone of the object.
- Must be a registry that maintains a cache of all the instances of the prototypical object

Each instance of the website has a default format, with default stocks and sidebar values inserted to begin. Each time a user opens a tab that accesses the website, it represents a unique instance. These instances are all originally cloned from the prototype, but change with regards to user input and preserved browser cookies. Each instance of the website will also be preserved between uses, remaining in the user's browser.

Appendix A - Function List

Some functions not relevant to architecture and design, e.g. testing functions, have been omitted.

Identifier	Name	Description
F1.3.2	pull_data()	makes a call to the API Function and returns data to terminal or file
F1.4.2	test_pull_speed()	makes a call to pull_data() and returns speed of 1 call, 5 calls, 10 calls, and 100 calls
F1.4.3	test_pull_efficiency()	makes a call to pull_data() and returns the number of calls that can be made in 30 seconds
F2.3	interpret_JSON()	takes the output of pull_data() and returns an array of values indexed according to data in JSON
F2.5	interpret_JSON_speed()	calls pull_data() => interpret_JSON() and finds speed of entire operation for 1 and 10 consecutive calls
F4.4.3	get_stock_price()	takes data from pull_stored_data() and returns the stock price for a given Ticker symbol
F4.4.4	get_stock_info()	takes data from pull_stored_data() and returns an array containing the stock info
F4.4.5	get_stock_news()	takes data from pull_stored_data() and returns two news articles in an array for a given Ticker symbol
F4.4.6	get_stock_info_time()	takes data from pull_stored_data() and returns the time of the latest update of the stock data
F4.4.7	get_stock_graph()	takes data from pull_stored_data() and returns a graph containing stock data for a given interval (1 day/week/month/year)
F4.4.8	get_stock_history()	takes data from pull_stored_data() and returns an array of data of stock data for a given interval (eg. 1 day, 1 week, 1 month, 1 year)
F4.5.1	make_stock_graph()	calls get_stock_history() and creates a graph using node points on fixed spacing for a given interval (1 day/week/month/year)
F4.5.2	make_stock_delta()	calls get_stock_price() and get_stock_info() to compare the market open and current price to return the delta (either as true or %)
F4.6.1	display_stock_price()	calls get_stock_price() and displays the data in the associated div
F4.6.2	display_stock_info()	calls get_stock_info() and runs subfunctions to interpret the data and format it for display purposes
F4.6.3	display_stock_news()	calls get_stock_news() and displays the data to the associated div flex-container
F4.6.4	display_stock_time()	calls get_stock_info_time() and displays the time of the last update to the associated div
F4.6.5	display_stock_graph()	either calls and displays get_stock_graph() or make_stock_graph() (mutually exclusive cases)
F4.6.6	display_stock_delta()	calls make_stock_delta() and displays the data in the associated div

F5.1.2	pull_ticker_list()	uses API functions to access a list of all supported Ticker Symbols
F5.1.3	push_ticker_list()	pushes the ticker symbol list to a database (generated once per session, or alternatively static)
F5.2.2	find_associated_tickers()	takes string input and returns an array of 5 suggested stocks by ticker symbol
F6.1.3	backend_run()	calls all necessary backend functions to initialize website
F6.1.4	frontend_run()	calls all necessary frontend functions to initialize website
F6.1.5	run()	calls both backend_run() and frontend_run() and start_clock()
F6.1.6	initialize_website()	calls all the getter scripts initially
F6.2.2	start_clock()	maintains an internal timer for the website instance
F6.2.3	get_clock_state()	returns the current time that the clock has been running