McMaster University – Bachelor of Technology

ENGTECH-4SA3 Software Architecture

“Money Growth” (Investment Assistant Software) Architecture Documentation

Student Name: Yihuan Zhang

Student Number: 400350335

Date submitted: July 26, 2021

Submitted to: Kevin Browne

Version: 1.3

- Added Development viewpoint, Logical viewpoint, Process viewpoint to the “4+1 View Model” section.

Description

In the project of this course, I am interested in designing a software that manages and outlines the user's wealth and investment, the software will be called "Money Growth". Unlike SimpleWealth that directly trades for users, "Money Growth" is a management system that records the cost and values for users and focus on analyzing which part of the portfolio is loosing money, which part is growing. Since investors often time invests their money on Simple Wealth, Stock market, bond, precious metal, crypto-currency, real-estate and many others, it is hard to track your total wealth growth across many different places. The "Money Growth" system takes data from both users' input and API from trading systems, and calculates the profit of each investment.

Target users of "Money Growth" are various of people that has already, or prepared to, invest their money in different ways. Both experienced investors and beginners can use it for tracking their own investment portfolios no matter how different their properties are.

In order to design such application, a typical 3-tier architecture can be used. A database contains user data should be implemented behind the presentation and application layer. Moreover, a publisher-subscriber style architecture will probably also be used when the software requires JSON message (e.g. stock value, gold price) through API (from other servers on the internet).

In order to initialize users' portfolio, Creational Pattern will also be used to load data. More specifically, a Factory Pattern class will be used to define the individual assets and their attributes. There may be some calculation or other logical computation involved in creation of objects, such as calculating the average cost, increasing or decreasing ratio, etc.

The technologies including API, OOP, SQL database, will be used to process and store the users' data, and Python or JavaScript can do the calculation (for instance, the proportion, average costs, relative profit, etc) and the presentation (either a web browser or through terminal). If time is permitted, a graphical interface and analysis will be an extra feature.

4+1 View Model

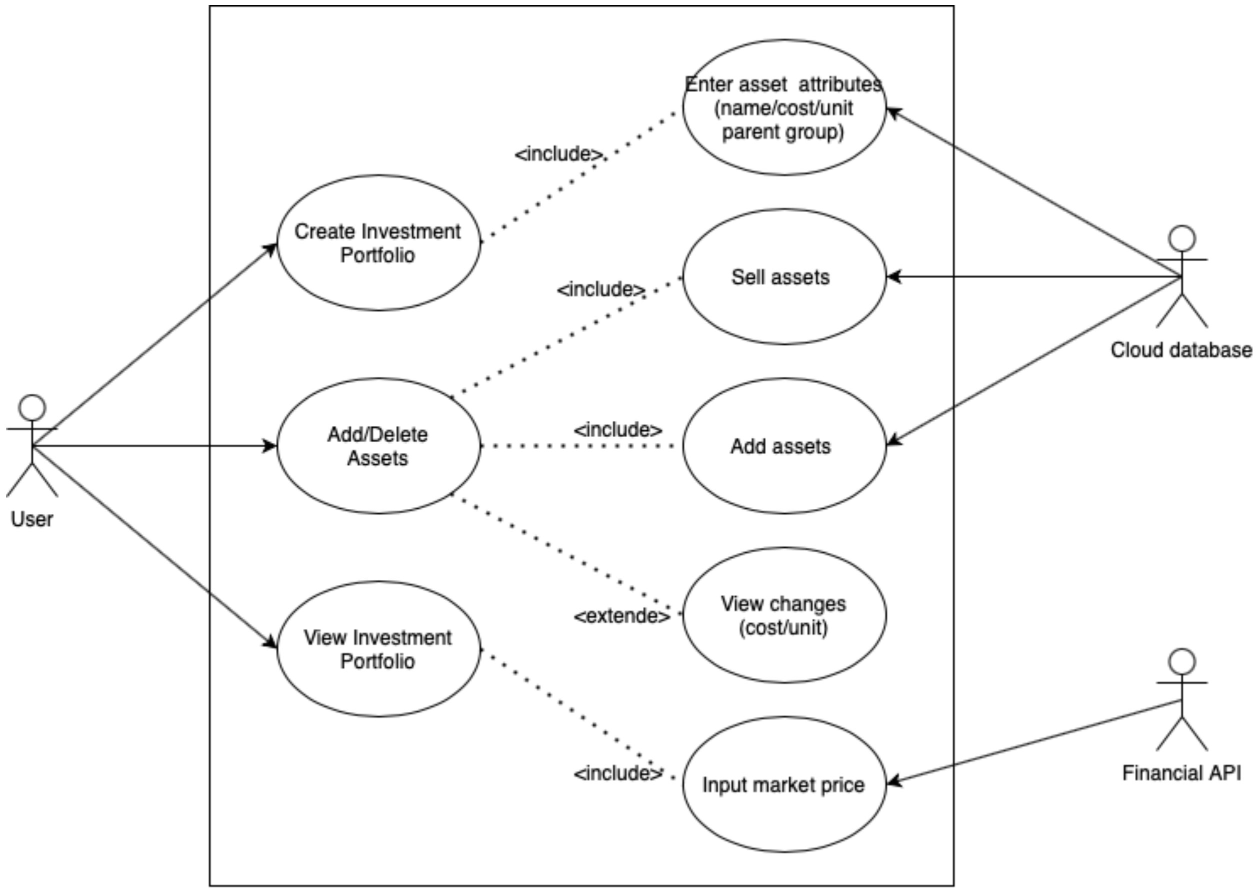
Requirements

	Functional Requirements
R01	Users shall be able to add their assets to the system
R02	Users shall be able to view the data they entered
R03	Users shall be able to make any change to their assets
R04	The system shall calculate and provide the cost, profit, and annual gain
R05	The system shall be able to retrieve the market data from both external API and user input

	Non-Functional Requirements
Performance	The system shall retrieve the data and initialize objects in less than 10 seconds
Availability	The system shall exceed 99.9% uptime and no more than 1 minutes of restarting time (MTTR)
Performance	The provided price shall up to date, with no more than 24 hours latency to the market price
Usability	The training time shall take no more than 10 minutes for a new user to learn to operate the system
Scalability	The system shall be allowed to adapt new database and interface

Scenario viewpoint

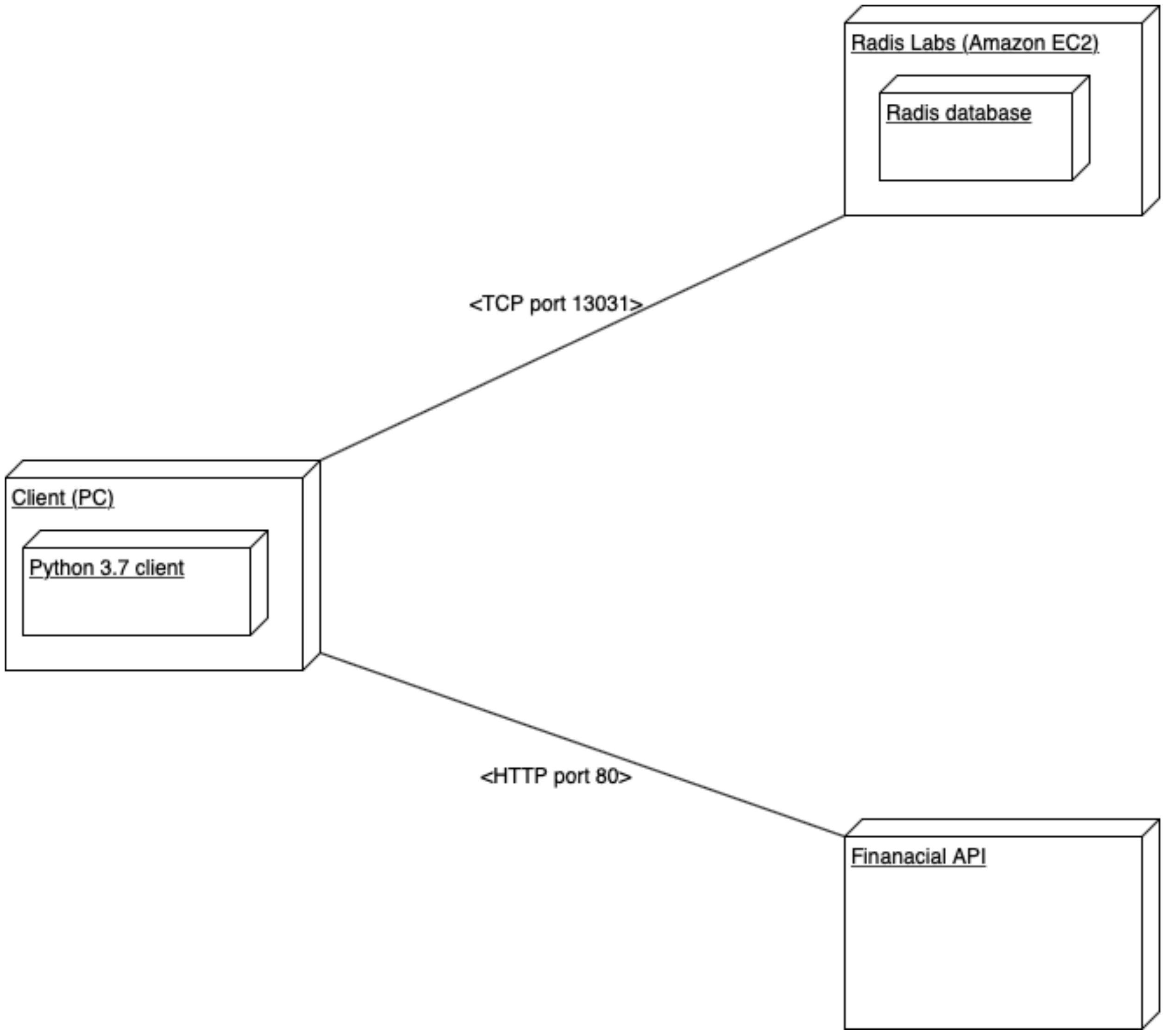
There are three scenarios that users use this software for: Create investment portfolio, Add or delete assets, and View investment portfolio. Usually, a user only need to create his portfolio once (at his first time of using the software). Then he can weekly or monthly view his portfolio performance by updating the market price (either by an external API or manually enter it), the increasing/decreasing ratios with relative profits (weight of total assets) will be calculated and show to the user. The user can also adjust his portfolio when he sold or bought more unit of his assets, users can also add new assets. See figure below for the Use Case diagram.



(Figure 1 - Use Case diagram)

Physical viewpoint

The software written with Python3.8 is installed on client PC, it communicates with the cloud NO-SQL database via TCP section at port 13031, user's assets' attributes will save in the database. When the user needs to view his investment portfolio, the system acquires the market data from a financial API and perform the calculation. The market prices are JSON data from external source access by HTTP, it will not save to the database. Refer the figure 2 for the deployment diagram.



(Figure 2 - Deployment diagram)

Development viewpoint

The software uses a file called “config.py” to determine some pre-configured read-only data, including API addresses (and database connection), only developers (or contributors) can edit this file when necessary maintenance/configuration needed. The Singleton pattern can be used here to read the configuration file and restrict the instantiation of the class to one object only but global accessible.

To create objects of different assets, creational patterns will be used. An abstract factory determines the interface of assets which contains methods such as:

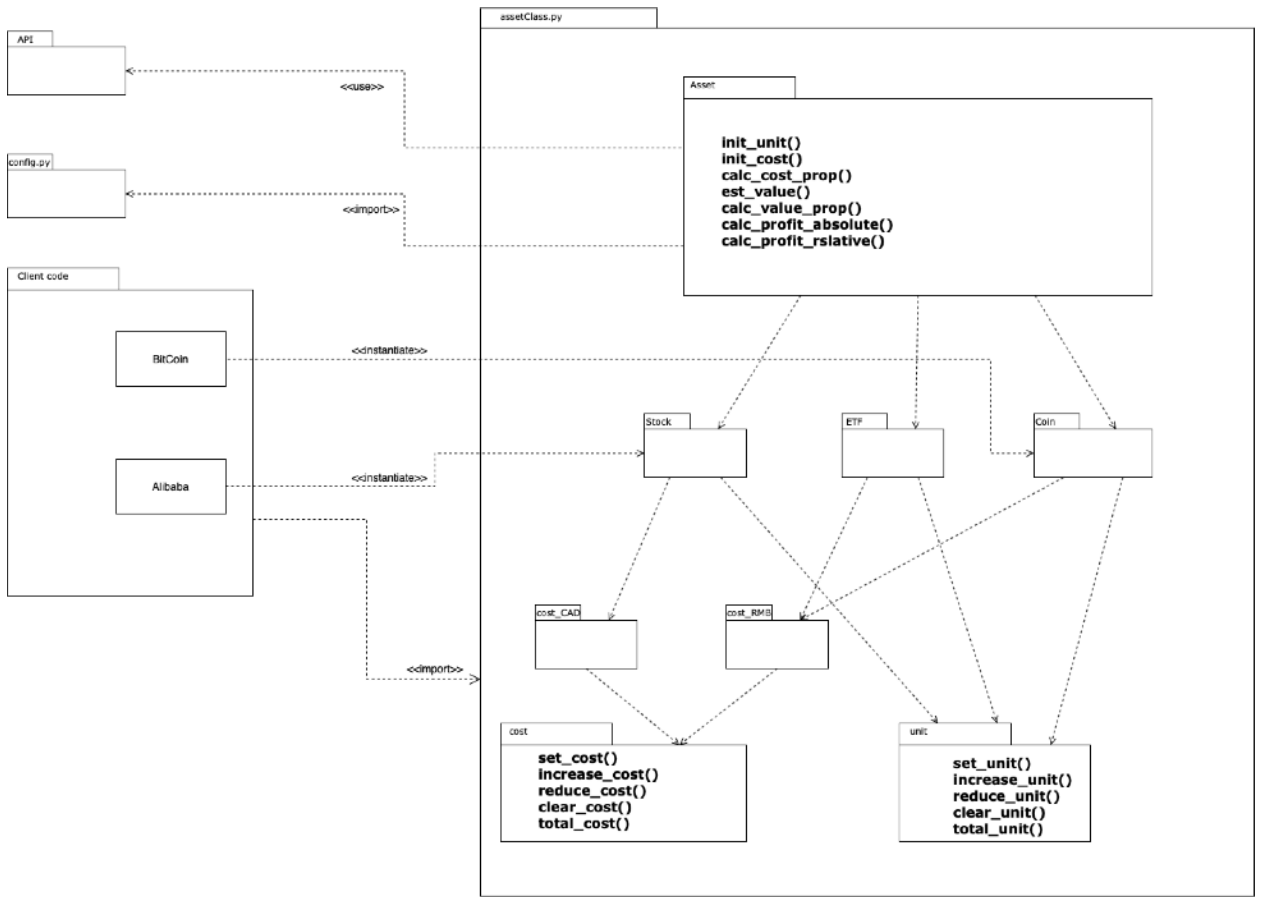
- “initializing unit”: initialize the amount of units when buying an asset
- “initializing cost”: initialize the amount of cost when buying an asset
- “calculating cost proportion”: calculate the cost proportion of an asset to the total
- “estimating value”: retrieve the market value of an asset via API
- “calculating value proportion”: calculate the market value of an asset to the total value
- “calculating absolute profit”: calculate the ratio of profit to the cost of a certain asset
- “calculating relative profit”: calculate the ratio of the proportion change to the value portion of a certain asset

Concrete classes will be portfolios (stock, ETF, CryptoCurrency) and inherit the above abstract factory. Each concrete class will have the attributes like unit and cost, but may use different APIs for estimating market value. Also, this part can be further scaled (i.e. different exchange rates conversion) according to the future demands (global assets or foreign users) without any client code changes.

In the client code, the objects are the assets inherit from the concrete classes. For example, in the UML diagram below the BitCoin is an object of the class “Coin”, which inherits the abstract class “Asset”, and uses “cost_RMB()” as “cost” type. Similarly, the “Alibaba” object is a “Stock”, it is also has all the methods in “Asset” and uses “cost_CAD()” as “cost” type. In other words, the BitCoin returns cost in RMB but the Alibaba returns in CAD in this case.

When requesting price data from the external APIs, a Command pattern can potentially be used to display the progress. If there are more than 10 different APIs to call, Concurrency pattern may also be used to increase the performance in the future.

The factory classes will be stored in a separate file, the rest classes (i.e. invoker, receiver) will be contained in the main code file.

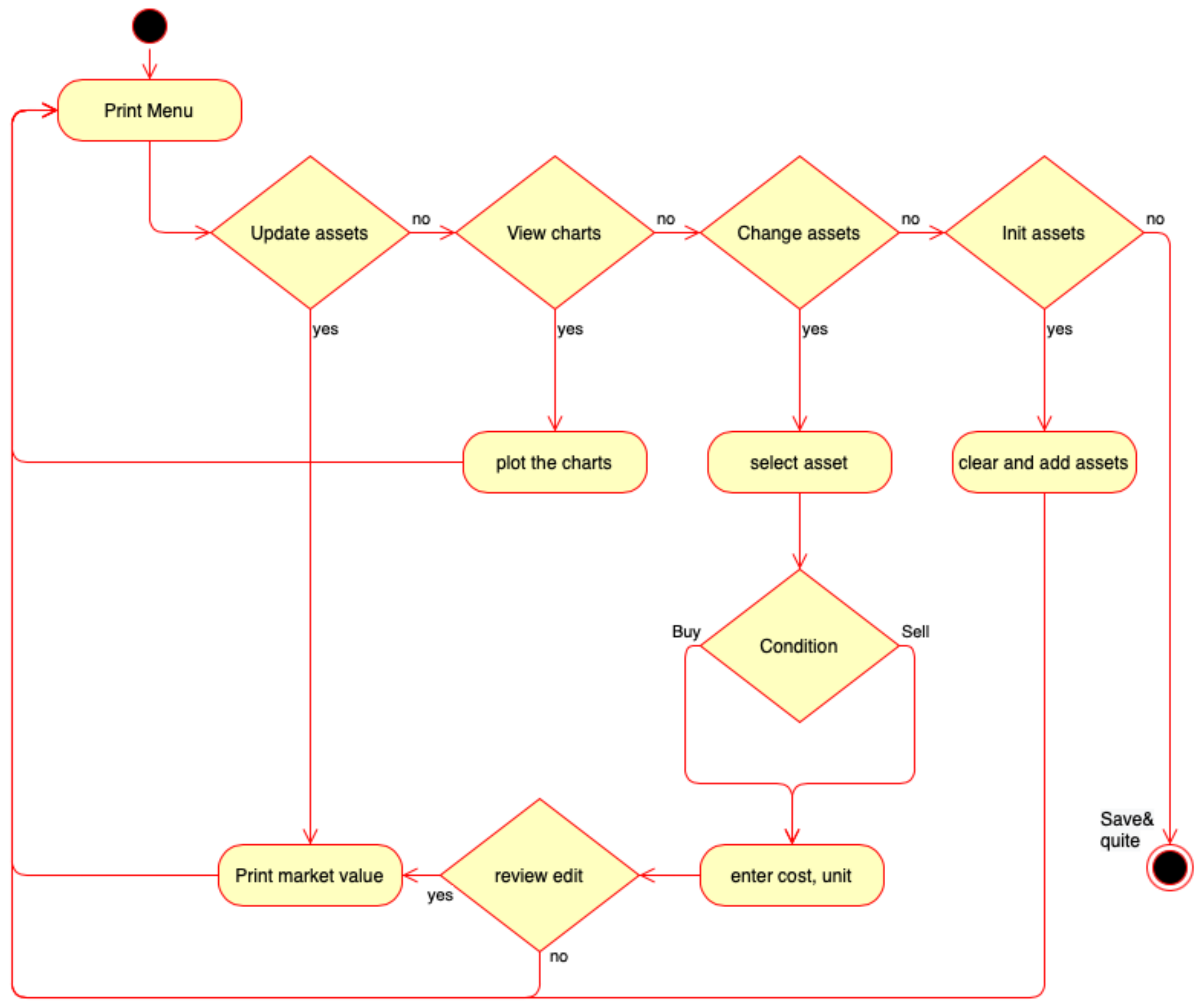


(Figure 3 - Package diagram)

Logical viewpoint

The logical viewpoint can be described as the UML activity diagram below. For the initial version of this program, the software uses command line as the interface and serving for the local user of the host PC. That means, the beta version of “Money Growth” will emphasize on asset management (instead of user management) and therefore will not have a login function, the end user will also need to type commands (instead of using graphical or web interface).

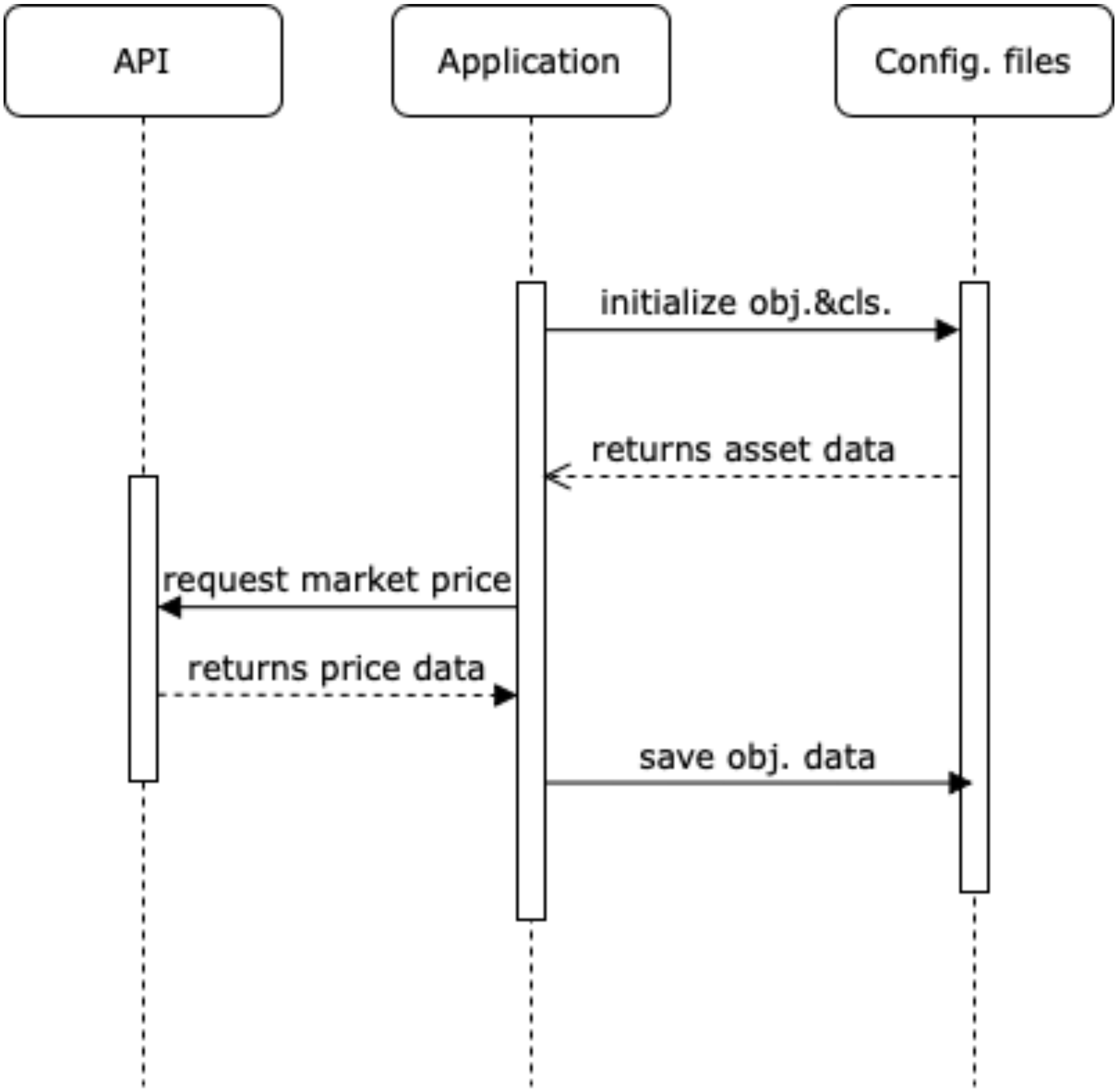
Briefly to explain, the user enters the selection according to the “Menu”. The “Update assets” calls the API to automatically updates market value and returns to the “Menu”; the “View charts” plots the proportion of the assets and returns to the “Menu”; the “Change assets” is used to edit the assets when a user bought/sold his assets which will lead the change of the cost and proportion, it will perform some input-check to prevent invalid data or overflow, and print the changed asset before saving the data; the “Init assets” is used to add many assets at the beginning by enter all the parameters, it can also be used to factory reset the software.



(Figure 4 - Activity diagram)

Process viewpoint

The UML sequence diagram illustrates how the “Money Growth” application interacts with APIs and Config files. The application firstly initialize with a class file that contains factories to instantiate asset objects, then it may also retrieve data from another file if it is not the first time of using it. Whenever the “Update assets” function being called, the application sends a request to the corresponding API and retrieve the price data from the internet. At the end of the process, the application can also write (or create) asset data to a file.



(Figure 5 - Sequence diagram)