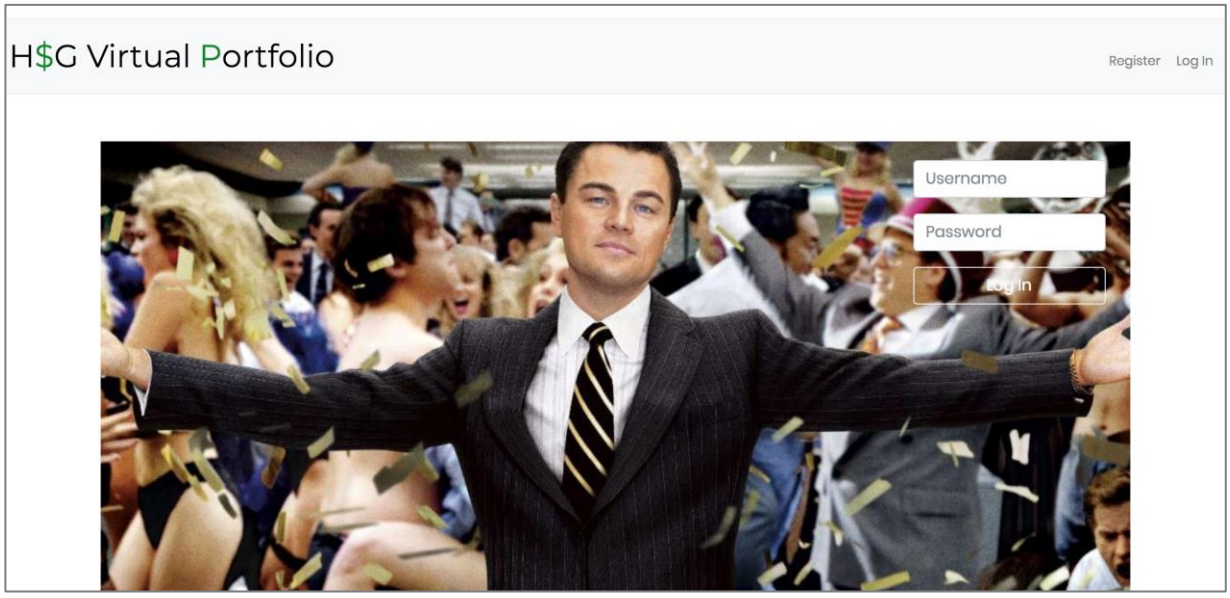
Universität St.Gallen

Business Administration, Economics, Law and International Affairs



H$G Virtual Portfolio

Diana Schoeller
Alexander von Stegmann
Maximilian Michel

Applications in Object-Oriented Programming and Databases
Johannes Binswanger & Ruben Zürcher
FS18-6,264,1.00
22. Mai 2018

Table of Content

Table of Figures

1. Targeted Use Case

It was our goal to develop an application that could serve as a practise investment platform for HSG students, i.e. a virtual portfolio. The idea was, that each user is given the same initial capital of CHF 10'000, which he / she is then free to invest in any listed stocks that are traded in USD, EUR or CHF. Firstly, we decided that the user should be able to register, login and logout. Within the application we aimed that he / she could quote stocks for their current prices (via Yahoo Finance API) and find out most recent news on the respective company (via News API or Bloomberg). Moreover, the application was meant to provide the opportunity to "buy" and "sell" stocks, track one's own portfolio, view all transactions conducted and compare the performance of one's investment strategy against that of fellow students.

As we had a clear use case and best-case functionality in mind, we started the project by drawing out the front end with all user interfaces and respective functions necessary on a blank sheet of paper. With most necessary functions gathered, we backwards engineered the backend structure. We drew out a large class structure and brainstormed on what class had to fulfil what functions and how they interacted with each other. Simultaneously, we started to design the data base structure, keeping in mind that it ought to be as normalized and lean as possible. After the conceptual phase, we split the work in line with the Model-View-Controller (MVC) logic into Front-End (View) and Back-End (Model, Controller). During individual work periods we kept in touch via messenger, and then merged and enhanced our progress in several longer working sessions. Unfortunately, we only discovered the usefulness of GitHub repository on the last mile of our project, but at least this is what we are using now for coordination and collaboration.

2. Application Overview

As mentioned above the underlying structure of the application is based on the *MVC* principle. At first this strict differentiation between the elements of the applications seemed like a large effort for a project of this size. Deeper into the development process however the benefits of such an outlay where clearly visible. Tracing back an error messages for example turned out to be much easier due to fact that it was clear where in the information chain the error occurred. This structure further allows for easy adaption to changing external circumstances as in deprecated APIs or database migration.

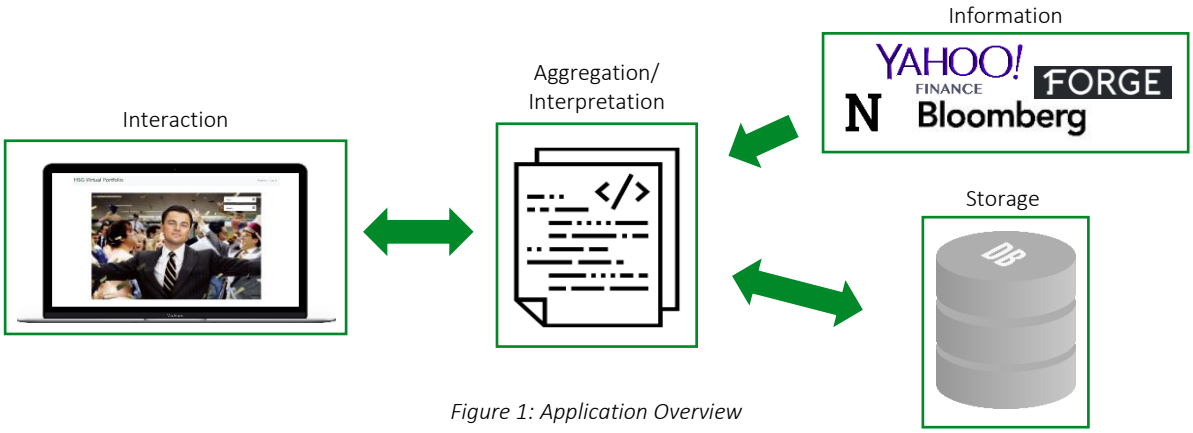


Figure 1: Application Overview

The view-component is represented by a Flask application that renders multiple html pages, which can then be accessed in a browser. Through the implementation of a sleek and easy to use user interface the interaction with our application is very intuitive and does not require pre-existing knowledge about the source code.

The Controller consists of an object-oriented class construct, that aggregates and interprets information from the connected APIs. This data is then stored in the Model and made available for user interaction. The model, i.e. our database, is separated from the rest of the application by a layer of data access objects (DAO), so that a change in the database-type only requires changes in this layer, but not in the whole application.

3. Backend

This chapter is going to further illustrate how the backend component of the application was designed and will explain our intentions behind the respective design choices. This description will mainly focus on aspects of object-oriented programming (OOP) and database design and how these principles were combined with the MVC approach.

3.1 Controller Structure

As centrepiece of the application, the controller requires to be thought through extensively. Following the OOP principles this process begins with identifying objects that exist in a portfolio. To make life easier, not just for ourselves but also anybody who wants to investigate the code, variables and function names carry are chosen to be self-explanatory. This description focuses on the small portion of elements where this might not be the case.

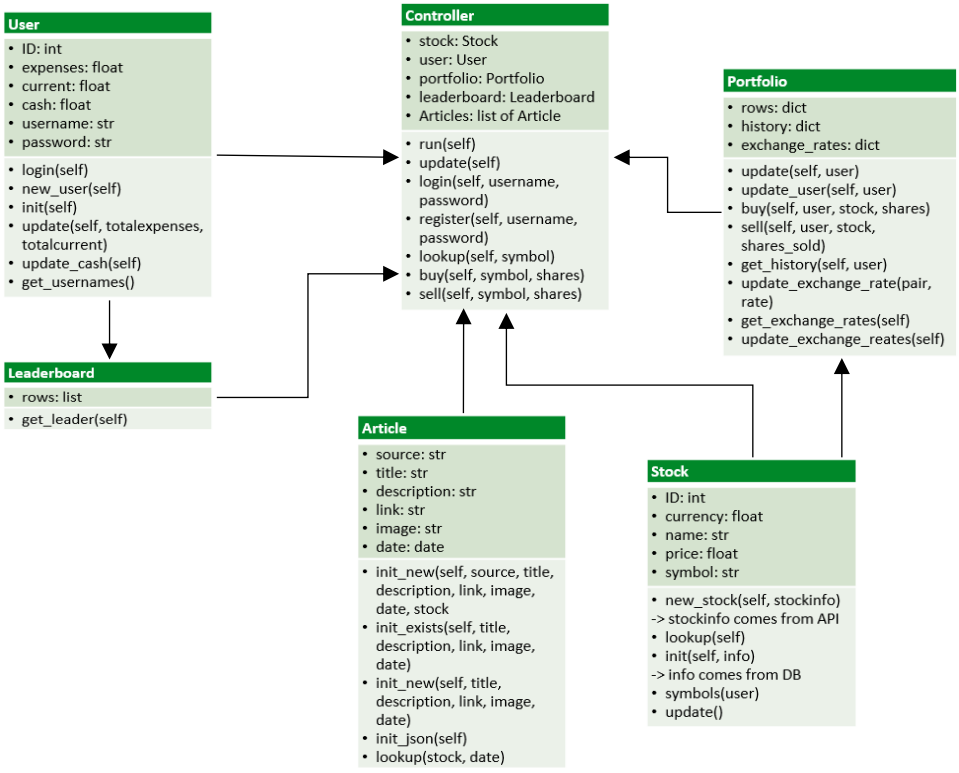


Figure 2 Controller Structure – classes based on OOP logic

When thinking about a portfolio, evidently the first object that comes to mind is the *stock* itself. As the class is used in many interactions with the database, where each stock is referenced by its ID, it was necessary to include the ID as an attribute. Besides more self-explanatory methods there are also the methods symbols(…) and update(). Symbols is a helper function for the view part of the application as its sole purpose lies in providing all the stock symbols of the stocks a user holds to create a dropdown list to choose from. The update method is used by a raspberry pi to update the stock prices on a regular schedule (on workdays from 9 to 22 to cover European and American markets).

The *portfolio* class builds on the stock class. The update method organizes all the stocks a user bought as well as additional information, such as exchange rate or the user's cash, in a dictionary. Key functionalities used to buy and sell stocks can also be found in the portfolio. The history dictionary in turn contains every transaction conducted by the user.
To be able to differentiate between diverse users the object *user* is needed. In our design, the class is not only used for the functions of logging in and out, instead it also carries the user's own cash and the current value of his portfolio. This design choice was made with the *leaderboard* functionality in mind.
To add further functionality, the class *article* was needed. It supplies the stock lookup function with articles that are either fetched from a news API or the database.
The objects mentioned so far are used to build up the core functions of the application. The *controller* class is finally introduced to hold the objects and make it possible to call functions, including several methods. For complex interactions between the user and the application the controller holds specific methods. E.g. if a buying process is initiated the method buy(…) in the controller makes sure that the user inputs passed fulfil certain criteria and that the required objects are initialized. For the most part, this is the class, which the view part of the application is going to interact with. It is also the only class that is initialized in the Flask application script.

3.2 Model Structure

As we aimed for the application to be accessible to multiple people at the same time, the database needed to be located on an online server. This setup allows for consistency of information across all user as the database is running 24/7 and can be updated independently.

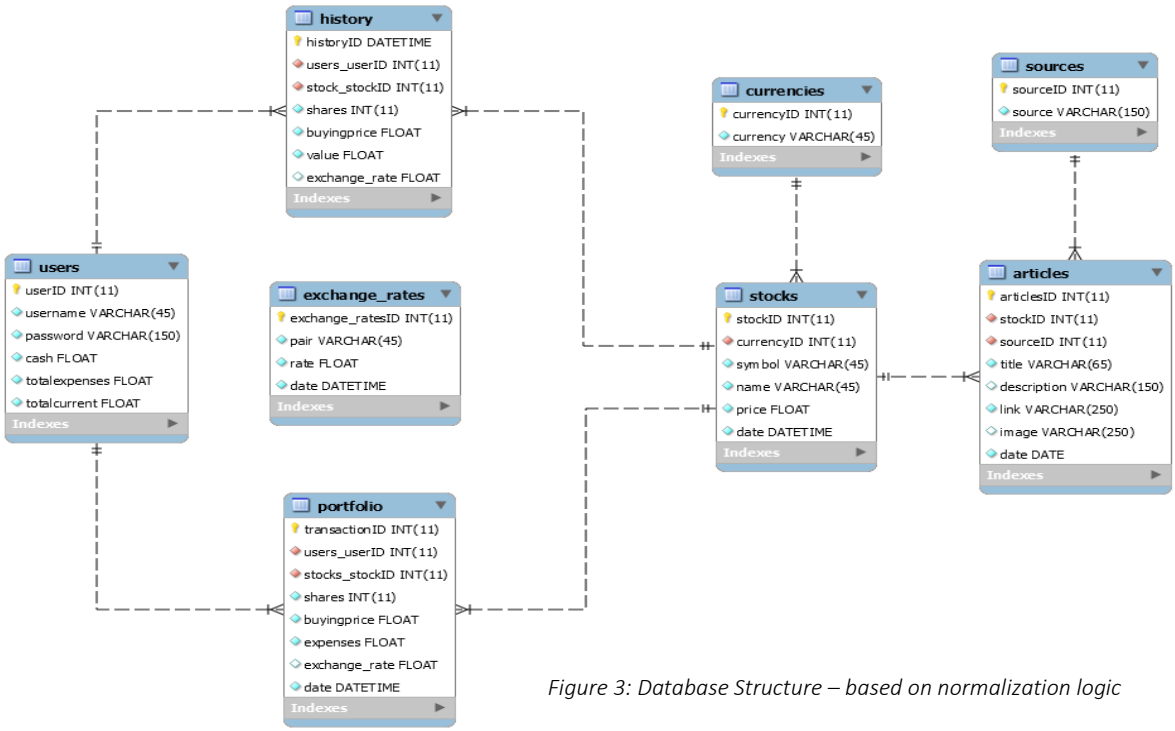


Figure 3: Database Structure – based on normalization logic

The tables *users* and *stocks* build the base of the database structure. These tables hold the most important information as the application would not run without a user or stock entries. Most of the other tables hold at least one of their keys as foreign key.
Where possible the rules of normalization where applied to avoid storing the same data entries multiple times. Normalizing the tables proved to be a very challenging part in the design phase of the project. Regarding the portfolio this process worked very well as the aggregated information is spread across

multiple tables eliminating redundancy. On the other hand, it needs to be pointed out that the architecture of the *article* table should be revised. In some cases, the same article is stored multiple times as it relates to different *stockIDs*. Due to time limitations this problem still exists. A possible solution would be the creation of a separate table, where articles are stored and then reference the respective key in the current article table to create a relation to *stockID* and *sourceID*.
Moreover, it is noteworthy that the *user* table stores the information about users' total values of cash, expenses and the overall portfolio value in addition to the common entries username and password.

3.3 API Interaction

Regarding the functionality of our application, we needed four APIs to acquire different sets of information. First and maybe most important in our case, we used the *Yahoo Finance API* for obtaining real time prices of the stocks that we aimed to be available to purchase in our application. This API is used whenever a user quotes the current price of a stock, buys or sells something. Since the Yahoo Finance API was changed recently, there is no updated documentation, which made the implementation of this API a time-consuming process including a substantial amount of trial and error.
Second, for calculating the user's portfolio and ranking correctly, we needed real time exchange rates. Those were accessed via the *1forge API*. While theoretically we could have implemented every existing currency, we decided on first only allowing stocks traded in EUR, CHF or USD. If we were to include further currencies into the application at a later point in time, the adaptations to the current code would be minimal.
Third, as an add-on to the quote function, we sought to offer the latest news available to a respective company, whenever a user queried a certain stock price. To achieve this the *News API* was used, which enabled us to search for articles by a keyword. Although it would be possible to display up to 25 articles per search, for clearness and convenience considerations, we decided on only showing the latest three news articles together with a link to the respective website.
Finally, our fourth API is the *Bloomberg API*. This one is used in case that there are no news available from the News API for a certain stock. This most likely happens in the early morning, especially due to the time difference between Europe and the US. The user is then told that there are no "News available on the given API" (News API) but is given the possibility to get redirected to the Bloomberg page.

In summary, four APIs were integrated to design our application in a way, that provides all information needed for calculations as well as creating the user experience we aimed for.

4. Frontend

To provide a more convenient, user-friendly and enjoyable user experience, the python micro framework *Flask* was used instead of the console. The application renders multiple html pages, which can then be accessed in a local browser. The interaction experience is essentially the same as on a common webpage. A function, for example login(), represents one interface and holds the interaction with the backend such as passing user input. This input is used to render either the desired information or an error message, derived from the apology() function in the helpers script, which is then shown to the user. To make error messages a bit more fun for the user, we converted them into memes based on an example found on GitHub. As we had high ambitions on the user interface, we faced the challenge of thoroughly understanding Flask, html, CSS and Jinja, which was a very time consuming, initially frustrating but finally rewarding experience. A tool that help a great deal in designing the front end, was *Bootstrap*, a free front-end framework that includes HTML and CSS based design templates for many website elements, such as buttons, tables, navigation and forms.

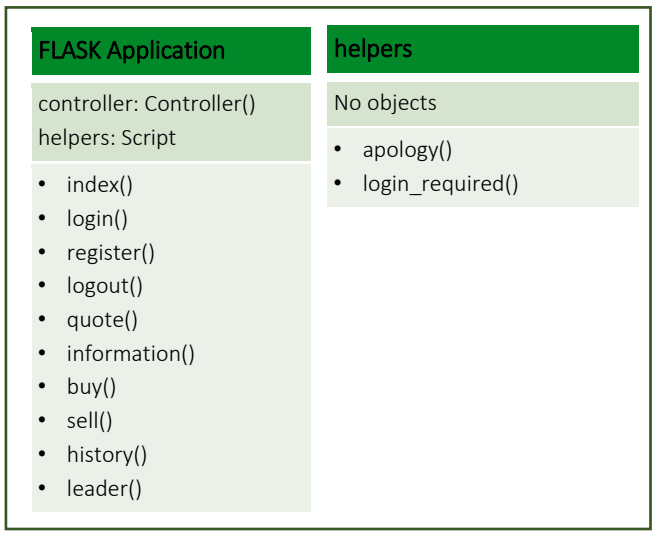


Figure 4: View Structure

5. Summary

Looking back at our target use case for this project, it is fair to say that we accomplished all our goals. This does not mean that the application is perfect. Some issues still remain, that we would have liked to tackle, e.g. the articles table in the DB, but as they do not affect the overall performance right now, we had to focus on the essential aspects of the application. As the learning curve throughout a project like this is very steep, it goes without saying that there are of course elements of the code or design choices that we would implement differently in hindsight. Overall we are very pleased with the outcome and believe to have learned a lot throughout this course. In case, someone is interested in taking a more detailed look into our code, this can be done by following the links located in the appendix.

Appendix

GitHub (This includes a readme):
https://github.com/avstegmann/HSGportfolio

Database access:

Server:	sql7.freemysqlhosting.net
Name:	sql7234823
Username:	sql7234823
Password:	WEhqj9sjiX
Port number:	3306

APIs:
https://finance.yahoo.com/
https://newsapi.org/
https://1forge.com/forex-data-api