VFTS DOCUMENTATION

(Virtual Fund Trading System)

Software Engineering

Professor Lihua Xu

Engineers: Amy Ao, Axl Cheng, Ningran Song, Zixiao Yang

21 May 2021

TABLE OF CONTENTS

1. DESCRIPTIONS

1.1 Background of the Project

During the past 2020, we've seen an increasing tendency among the Chinese college students and recently graduated workplace newbies to try investing in funds. One evident fact was that most of them were investing their living expenses or limited pocket money, which means that the amount was relatively much lower. They were just seeking to feel the trading environment and attempted wealth management to gain more experiences to learn from as the empirical.

Therefore, we designed this virtual fund trading system, which aims to provide the users with a mock trading environment where they can practice fund investment skills and accelerate related experience. Real-time data is applied in it to simulate a realistic environment, where users can trade funds using virtual assets associated with their account.

1.2 Introduction of the Software

VFTS is a platform imitating some real trade ones like *Ant Fortune, Tencent Financial Management, Tian Tian Fund* and so on. It provides some basic functionalities for fund trade processes including account register and login, update account security information, search funds' information, query the information of a certain fund in details, place and withdraw orders, add and remove self-listed funds, and also check account trade history and assets' information.

We developed *VFTS* in three parts, as front-end, back-end and database. Front-end is built with *Vue.js* framework while back-end with *Spring Boot*. For the database, we used *MySql*. There are more tools and packages we utilize to complete this project, which will be introduced later in the following sections.

2. DEVELOPMENT PROCESS

2.1 Planned Process

The eXtreme Programming is used in our development process with iterations in which we plan to complete certain user cases every two weeks. After completing UML designs, we started with our database design. Then we proceeded to build the front-end and back-end at the same time. During back-end development, unit tests are also implemented to test the performance of completed interface and service layers in the back-end. When all the unit tests are passed and controller layers also well done, we then go to joint debugging and test whether requests and responses can be successfully sent between front and back ends.

2.2 Iteration Tasks Specifications

Iteration 1 (MAR 22 - APR 4)
- Design and Implement the Database
- Build the framework of server
- Build the framework of front-end

Iteration 2 (APR 5 - APR 18)
- UC 1 Log in
- UC 6 Update Account Security Info
- UC 3 Query Fund Information

Iteration 3 (APR 19 - MAY 2)
- UC 2 Learn Basic Fund Information
- UC 4 Trade Funds
- UC 5 Check Personal Account

Iteration 4 (MAY 3 - MAY 21)
- Unit Tests
- Joint Debugging

3. REQUIREMENTS & SPECIFICATIONS

3.1 Use Cases

We implemented 6 use cases in total in the software as following:

UC1 Log In

1.1 Preconditions: None.
1.2 Main Flow: Users enter the username and login password to log in.
1.3 Subflows:
[S1] If users forget their login password, they can reset it through security question answers.
[S2] Remind users when their login passwords do not match with usernames.
1.4 Alternative Flows:
[E1] New users register with username, login password and other information (payment password and security question answer).

UC2 Learn Basic Fund Information

2.1 Preconditions: None.
2.2 Main Flow: Users can learn some basic fund knowledge
2.3 Subflows: None.
2.4 Alternative Flows: None.

UC3 Query Fund Information

3.1 Preconditions: None; further operations such as adding certain funds to the self-selected list requires login.
3.2 Main Flow: A list of funds is displayed with the real-time quotes and accompanying information
3.3 Subflows:
[S1] Users without logging in are allowed to browse the list and click on any fund to view the detailed information.
[S2] Search for a certain fund with fund ID.
[S3] After logging in, users are further allowed to add certain funds to the self-selected list.
[S4] After logging in, users are allowed to buy certain funds [UC4].
3.4 Alternative Flows:
[E1] If users attempt to add to the self-selected list or to buy funds without having logged in, they will be prompted to log in or create an account [UC1].

UC4 Trade Funds

4.1 Preconditions: Users select a particular fund through either UC3 or UC5.
4.2 Main Flow: Users can either make an order to purchase or redeem the selected fund. A payment password is required for the transaction. A reminder message of transaction laws will appear.
4.3 Subflows:
[S1] If users want to purchase the fund, they need to enter the amount of balance they want to invest and the payment password.
[S2] If users want to redeem the fund, they need to enter the amount of shares they want to sell out and the payment password.

4.4 Alternative Flows: None.

UC5 Check Personal Account

5.1 Preconditions: Users have successfully logged into the account.
5.2 Main Flow: Users can see the current total assets and the information of the funds they are holding.
5.3 Subflows:
[S1] Users can see the self-selected list of funds.
[S2] Users can trade [UC4].
[S3] Users can check the yields information of their accounts
[S4] Users can withdraw their processing orders, which are not processed to trade records yet.
5.4 Alternative Flows: None.

UC6 Update Account Security Information

6.1 Preconditions: Users have successfully logged into the account.
6.2 Main Flow: Users can update their account security information, such as login password, payment password and security question answer.
6.3 Subflows:
[S1] Users can reset their login password when they enter the right old password.
[S2] Users can reset their payment password when they enter the right old payment password.
[S3] Users can reset their security question answer when they enter the right old answer.
6.4 Alternative Flows: None.

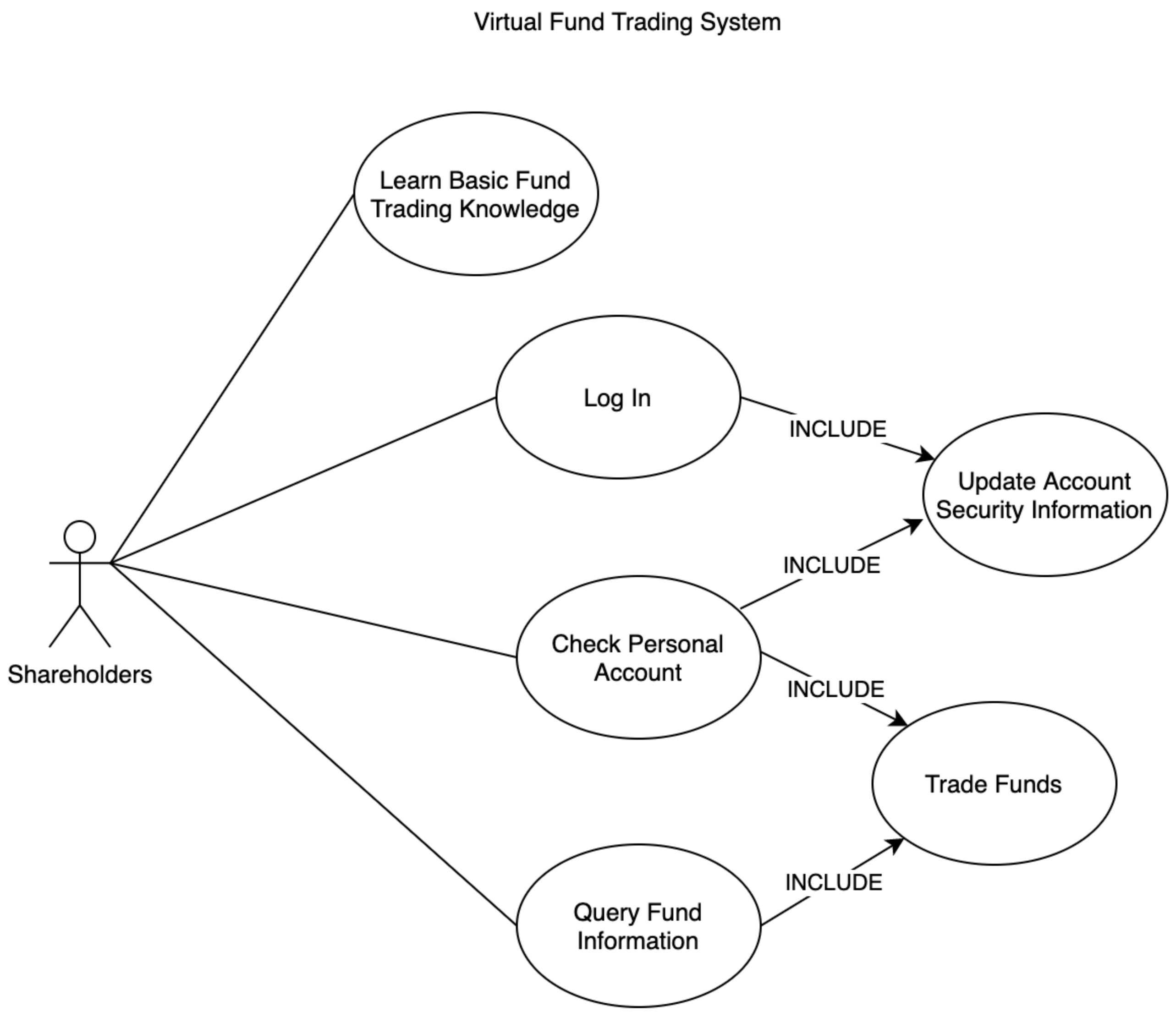
3.2 User Stories

All of our user stories have been stated in the use cases' main flows or subflows already. As mentioned above, we have 6 main functionalities: 1) account register and login, 2) update account security information, 3) search funds' information, query the information of a certain fund in details, 4) place and withdraw orders, 5) add and remove self-listed funds, 6) check account trade history and assets' information. These are the main requirements for a fund trade system.

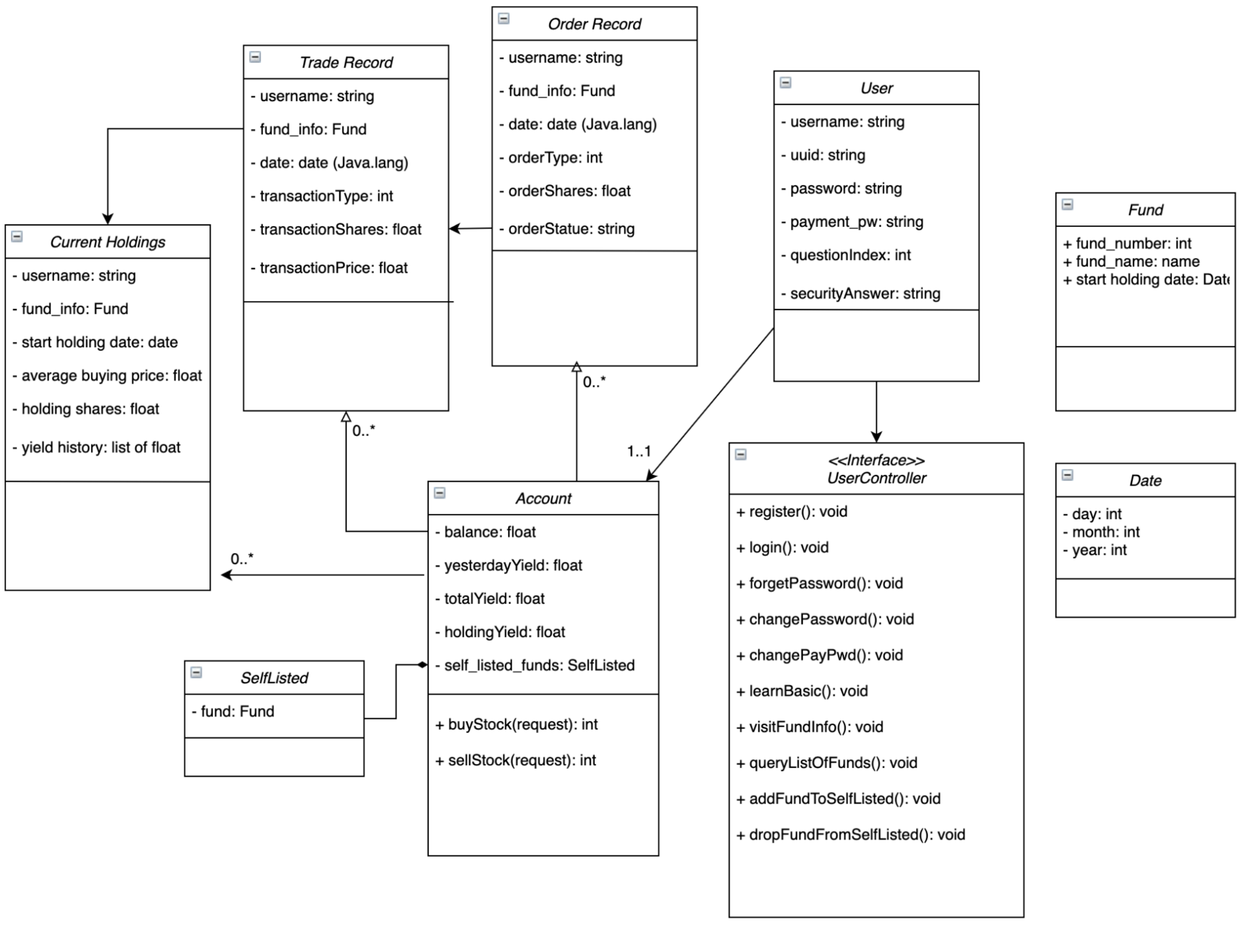
4. ARCHITECTURE & DESIGN

4.1 UML Designs

Use Case Diagram

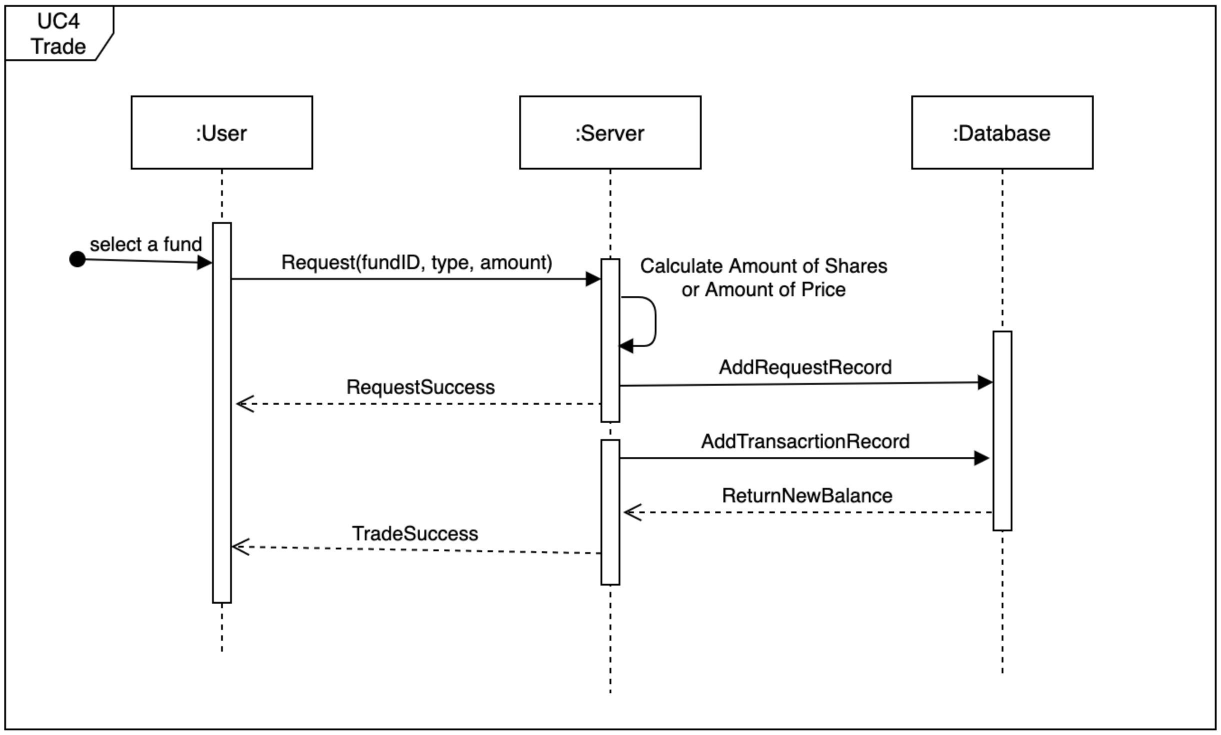


Class Diagram



Sequential Diagram

Trading is the core component of our system. This is the sequential diagram for a trade: A user can place an order to purchase or redeem a fund. The server processes the request by querying the database and making judgement whether or not the request is valid. After some time, the processing orders will be turned into a trade automatically, with the assets and yield updated.



4.2 Architecture

We follow the MVC pattern with *Spring Boot* framework in our project. Users can visualize the account information through the View, make requests through the Controller, and our Model will do the algorithm work and edit data in the database. The structure of the framework helps us to realize the MVC pattern. For instance, the *-server* part in the framework corresponds to the Model, the *-rest* part corresponds to the Controller.

4.3 Details of the Design

1) Front-end. For the view side, we implemented it with the *MVVM* model, as *Model-View-ViewModel*. We used *Vue.js* Framework to separate the whole front-end view into several components, including navigation bar, fund info page, fund list page, account info page and so on. Through combining the view showing all the data with *Vue's ViewModel*, we achieved reactive programming when the Model (Back-end) sends back responses to change the shown data. The front-end will automatically rerender the view side after sending requests and getting responses.

2) Back-end. Based on the use cases, we divide the back-end into two modules (user and trade), each implementing the corresponding functions. In each module, we have *-api* folder with Entity and Interface definitions, *-rest* folder with Controller, and *-service* folder with Mapper files and Service files. The user module implements functions like register, login, update passwords, forget passwords, and update security questions. The trade module implements functions such as add and show self-selected funds, buy funds, sell funds, show transactions, and so on. In particular, for the fund purchase function, we use timing to set the time of transferring orders into trades with the newest fund price every day.

3) Database. The database is implemented using *MySql*. *Mybatis* mappers are applied to connect to the server for processing queries or edits to the database. We include some tables for easier computations on different kinds of yield, especially the holding yield and the yesterday yield.

4.4 Framework Influences

The system helps us understand the *MVC* architecture during our implementation. It gives us a model to follow for connections of different parts.

5. REFLECTIONS & LESSONS LEARNED

5.1 Reflections on the project and process

Front End. There are lots of *api* works for our front-end to do besides the ones interacting with our own back-end. Because we don't tend to save all the fund info, we have to request many third-party apis to get the information and then handle all the data. One more obstacle for us is to visualize all the data of a fund or account. We utilized both visualization packages like *ECharts* and also <Span> to help render different data into different styles.

Back End: Since we all lack prior experience of such software developments, it takes us plenty of time to follow the structure, and we have dealt with many nested bugs caused by default properties of the framework.

Database. The setup of the database is crucial, because changes made on the data structures may result in many subsequent changes of the server commands. We have a complex data structure and computations. Group members need to be very certain about the data structures before making implementations to reduce most of the annoying changes.

5.2 Lessons Learned

Unit tests are very important for our development. We have many nested bugs during our implementations, and unit tests ensure that our structures and algorithms are on the right track. It saves plenty of time when we proceed on the joint-debugging.

We underestimate largely the time we need for a final compile. We have no prior programming experience with the *Spring Boot* structure and the maven is not at all intelligent. A slight difference in JDK version or a lack of certain dependency will cause the error of the whole project. We spent a lot of time on compiling the project, but we proceeded quickly once we compiled successfully. We will plan more time for the compiling process in our future programming experience.