An Open-Source Online Trading Platform

Danping Zhou

Master of Science

School of Informatics

University of Edinburgh

2017

Abstract

In recent years, online trading platform become increasingly popular for financial products, such as Bitcoin. However, most existed online trading platforms are close-source, and normally are constructed with traditional developing languages and technologies, which possess higher security risks. This thesis aims to build a self-developed Open-source online trading platform for Bitcoin based on advanced functional language Scala and integrated with a number of novel technologies, including MongoDB, Play Framework, AnjularJS.

The full developing process are documented from demand analysis to front-end and back-end implementation. The system requirements extracted from demand analysis provide guidelines for initial system architecture based on classical MVC pattern. Account Management, Finance Management and Trading Management are three core modules for Front-end design. Numerous interfaces for different services are implemented through HTML, CSS and AngularJS. Rigorous validations are designed to improve user experience and avoid system errors. All required functionalities for this web application are successfully realized through Back-end implementation. A series of Data models are built to interact with MongoDB to manipulate data efficiently and precisely. Basic functionalities for different service modules are achieved by corresponded Controller. The Auto-trading engine, which is the most critical part for a trading platform, is optimized by numerous design iterations to achieve acceptable performance in terms of high efficient and accuracy for large amount of trading requests.

The evaluation results confirmed the efficient and accurate performance of Front-end and Back-end implementation. Particularly, owing to the parallel processing ability of Play framework, the system can correctly handle concurrent HTTP requests with high throughput (380 requests/s). According to the evaluation of Auto-trading engine, the concurrent issues have been perfectly avoided by sequential timing schedule job design. However, this system has to compromise on processing time when excessive trading orders are placed. In summary, this online trading platform for Bitcoin is successfully developed and integrated with most core functions. The combination of selected developing language and technologies are proved to be feasible and efficient architecture for trading web applications. This work can be used for future expanding to be applied in real world.

Acknowledgements

I would like to thank my supervisor, Dr DUBACH Christophe, for providing objective and useful feedback on my work throughout this project. He was very supportive in helping me to analyze the problems I encountered during the design process.

I would also like to thank my IRP tutor Dr Dave Cochran and IRR tutor Mr Yang LIU for helping me to make initial steps during my Masters degree.

Finally, I would like to thank my family and friends for their continues support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Danping Zhou*)

Table of Contents

1 Introduction **1**

1.1 Online trading platform 1

1.2 Why has Bitcoin become so popular 1

1.3 Motivation 2

1.4 Contribution 3

1.5 Organisation 3

2 Background **5**

2.1 Technologies 5

2.1.1 Scala 5

2.1.2 Play Framework 5

2.1.3 AngularJS 6

2.1.4 MongoDB 7

2.1.5 Apache JMeter 7

3 Project Overview **8**

3.1 Demand analysis 9

3.2 Constraints analysis 10

3.3 System Architecture 11

3.4 Summary 13

4 Front-End Design and Implementation **14**

4.1 Account Management Interfaces 14

4.1.1 Register Interface 15

4.1.2 Login Interface 16

4.1.3 Password Resetting 18

4.2 Finance Management Interfaces 20

4.2.1 Account Balance Interface 20
4.2.2 GBP Deposit/Withdraw Interface 21
4.2.3 Transfer Records Searching Interface 23
4.3 Trading Management Interfaces 24
4.3.1 Bitcoin Trading Interface 24
4.3.2 Trading Cancellation Interface 26
4.3.3 Orders Searching Interface 28
4.4 Summary 29

5 Back-End Design and Implementation **30**
5.1 Data Schema Design 30
5.1.1 Requirement Analysis 31
5.1.2 Conceptual Schema Design 32
5.1.3 Implementation Design 33
5.2 Basic Functionalities Design 34
5.2.1 Account Management Service 34
5.2.2 Finance Management Service 36
5.2.3 Trading Management Service 39
5.3 Auto-trading Engine 43
5.3.1 Business Requirements 43
5.3.2 Implementation of Auto-trading 44
5.4 Summary 46

6 Evaluation and Results **47**
6.1 Usability Evaluation and Results 47
6.1.1 UI Usability Testing 47
6.1.2 Basic functionalities Testing 49
6.2 Performance Evaluation and Results 50
6.2.1 Response time for HTTP Requests 50
6.2.2 Processing Ability of Auto-trading Engine 54
6.3 Summary 56

7 Conclusions **57**
7.1 Summary 57
7.2 Critical Analysis 58
7.3 Future Work 59

Bibliography **60**

Chapter 1

Introduction

1.1 Online trading platform

Online trading platform, also called electronic trading platform, is a software application developed to exchange financial products on Internet, such as stocks, currencies, gold etc. After the 1970s, a large amount of transactions have moved to online trading platforms, owing to its various advantages over traditional trading methods [16].

By these online platforms, users can purchase or sell a variety of products at any locations from all over the world. In contrast to traditional trading methods, online trading platform can display market prices for users in real time and provide additional multiple trading tools such as account management services. Moreover, users can submit trading requests and monitor the trading process at any time they want without any time restrictions. Another huge advantage is the lower fees comparing to other traditional trading methods, which contributes to an fast increasing number of users.

Among these financial products, Bitcoin is one of the most popular electronic currencies on these online trading platforms, and the demand of Bitcoin is continually growing in recent years.

1.2 Why has Bitcoin become so popular

Electronic payment has become the main payment method for the E-Commerce. It relies on financial institutions serving as trusted third parities to complete transac-

tions. Although most transactions can be processed correctly in this way, it still suffers from the inherent weakness of the based third parties. Since the financial institutions cannot avoid disputation mediation, the completely non-reversible transactions are unachievable. In addition, the costs of transaction are increased due to the costs of mediation, and the minimum practical transaction size is limited as well. According to Satoshi Nakamoto[17], these limitations can be prevented by processing payments without a trusted third party. In order to accomplish decentralized transaction on the E-Commerce, an electronic payment system Bitcoin based on cryptographic proof instead of trust was firstly invented in 2008.

Bitcoin is a decentralized electronic system built with peer-to-peer architecture. It allows direct transactions between two users without a third party. It can perfectly avoid the weakness of traditional electronic payment which is based on financial organizations. Furthermore, Bitcoin is very safe as it provides digital signatures to protect the ownership[5] .

1.3 Motivation

Owing to above advantages, Bitcoin has gained enormous popularity in E-commerce.[6]. With the increasing demands of Bitcoin, a number of online trading platforms have introduced the Bitcoin transaction functionality, and have attracted numerous customers to complete Bitcoin transactions on their platforms.

Nevertheless, most widely used Bitcoin online trading platforms such as Bitstamp, Coinbas and BitFinex are not open source. Open-source software is more secured comparing to close source, as the code is continuously analyzed and debugged by a large non-profit community. Moreover, the traditional developing languages and technologies implemented in these platforms have been considered as their weakness and limitations. For instance, PHP was regarded as one possible reason for the failure of Mt. Gox (a Bitcoin online trading platform), which used to dominate the Bitcoin trading market on the Internet. Mt. Gox was started in 2010, and then obtained a huge success, occupied 70% market for Bitcoin trading in two years[3]. However, it was suddenly bankrupted in 2014 after an hacker attack, which result to 850,000 Bitcoins disappeared (worth around half billion dollars)[2]. The bug of PHP architecture applied in its system was supposed to be used by hackers to attack. PHP is a server-side

scripting language without safety net, leading to inferior fault detecting ability [19]. Consequently, the system based on PHP architecture possess higher security risks. Similar to PHP, many other developing languages in some existed trading platforms like Python, Ruby and Node.js do not have great security performance either [20].

In light of this, there is a huge demand for an open-source online trading platform implemented with advanced functional programming language and high-performance technologies. As Scala is specialized in security, it is appropriate for trading platform development. In addition, other advanced technologies including MongoDB, Play framework are selected to build this trading platform after rigorous consideration. Given the great popularity of Bitcoin, it will be the implemented financial product for this project.

1.4 Contribution

A successful implementation of an online trading platform with essential functions would be the main outcome of this project. Meanwhile, the feasibility and performance of a series of novel technologies for this online trading platform will be explored for the first time through practical implementation and rigorous tests. Specifically, the combination of Scala, MongoDB and Play Framework for the software implementation have been proved to be highly efficient for large amount of HTTP requests and trading orders.

As one of the major motivation, this fully self-developed online trading platform will be open source and can be expanded with more practical functions to be able to apply in real world in the future. In this project, Bitcoin was selected as the financial product, while other financial products are also applicable for this online trading platform.

1.5 Organisation

This thesis presents the full design and implementing process of the online trading platform for Bitcoin, including demand analysis, design & implementation and evaluation. The development of each functionality on the platform requires rigorous analysis, elaborate design and scientific evaluation. The basic demand analysis is conducted

before starting the design stage. The requirements of target users should be considered throughout the design process and implementation. All these developing steps are described in details in this paper.

In Chapter 2, the detail review of several technologies applied in this project are introduced.

Chapter 3 introduces the Pre-design stage of this online trading platform. It provides an overview of functionalities that an online trading platform requires. The constraints of this project are also discussed. In addition, the whole system architecture is presented in this chapter.

Chapter 4 introduces the detail Front-end design process. Each interface has been introduced from different aspects, including functions, elements, implementations and actions.

Chapter 5 shows the design process of back-end of this platform, as well as the implementation. Data schema, basic functionalities and Auto-trading engine are introduced separately.

Chapter 6 presents the evaluation process and results analysis. Different kinds of testing methods for the performance evaluation of this platform are illustrated. A critical analysis is also included in this section.

At the end of this thesis, a final summary, a critical discussion of this project and its potential extension work will be given in Chapter 6.

Chapter 2

Background

In this chapter, several technologies that are proposed to be applied in this project have been discussed in detail.

2.1 Technologies

This project adopts Play Framework + AngularJS + Scala + Mongodb as its Technology Architecture, and the Apache JMeter is applied for system testing. The following sub-sections discuss the advantages and feasibility of each technology.

2.1.1 Scala

Scala is a functional programming language, and runs on Java virtual machine [12]. It features in extremely concise, increased productivity and extraordinarily powerful. An embedded library called Akka is offered to build asynchronous type-safe systems. Moreover, Scala provides various functions focused on security issues to protect the developed system from hacking. As a result, it perfectly suits the requirement for online trading platform[1].

2.1.2 Play Framework

Play Framework is a popular software application framework, written in Scala [13]. It can be integrated with Scala perfectly. The architecture of Play Framework can be

summarized as the following figure 2.1.

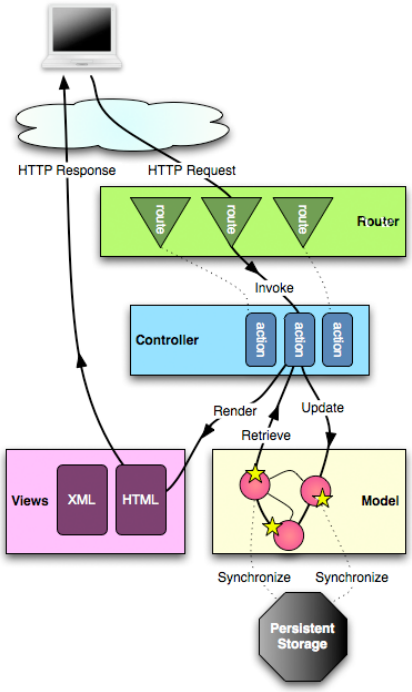

Figure 2.1: The architecture of Play Framework.(Adapted from www.playframework.com)

This software framework also support assets compilers, JSON and WebSocket. Applications written by Play Framework will do more jobs with the same hardware resources, because Play is an asynchronous, non-blocking HTTP server, which means the requests can be processed in parallel and waiting time is not needed.

Moreover, Play Framework has outstanding performance in terms of usability and maintenance. Comparing to CakePHP and Zend Framework, Play performs excellently in maintenance. All libraries used in projects can be downloaded automatically by Scala Build Tool(SBT), and no need to install any extra package on the server. According to these benefits, Play Framework is a good choice to be the framework of this project which can help to build a scalable and fast online trading platform.

2.1.3 AngularJS

AngularJS is a full featured front-end JavaScript framework which can be added to any HTML page simply with a script tag. The main goal of AngularJS is to simplify

the process of software applications development[15]. It supports the classic Model-View-Controller (MVC) programming structure and performs excellently at building dynamic, single page web applications (SPAS)[11]. In addition, AngularJS can handle heavy user interactions via forms with high speed. Since its ability to grab data and use it directly in makeup, AngularJS has a good performance in data visualization. Considering these features, AngularJS was applied to the front-end development of this trading platform, especially for the implementation of a dashboard which can display the real-time transaction records.

2.1.4 MongoDB

MongoDB is a free NoSQL database program invented by MongoDB Inc[7]. It has high flexibility and is capable of dealing with scalable issues easily. As it is document-oriented, it support data partition to manipulate data on multiple servers for big data problems. Both structured and unstructured data can be stored in MongoDB. Additionally, the MongoDB is schema-free, which means no schema migration would happen during software development. Most importantly, for high traffic web applications such as online trading platform, MongoDB can provides sufficient Back-end storage [9].

2.1.5 Apache JMeter

Apache JMeter, is an open-source software which is designed to do load tests and performance evaluation for various software applications[4]. It is capable of unit-test and functional test for various propose, including FTP, HTTP, TCP and DB connection. In addition, JMeter allows concurrent sampling by multiple threads to simulate a heavy load on a server and reflect the performance of system by generating a variety types of reports. Considering these features of JMeter, I planned to use JMeter as the testing tool of this project.

Chapter 3

Project Overview

This chapter gives an overview introduction of this project. The online trading platform for Bitcoin will be developed following typical developing process for web application. As represented in the classic Waterfall methodology (Figure 3.1), this project will start from demand analysis to identify system requirements. The constraints analysis work is another crucial part of the early development stage, as it will analyse all possible constraints that can influence the initial goals and balance the relationship between goals and constraints. After all system requirements have been clearly identified from demand and constraints analysis, an initial overall system architecture will be firstly proposed. The demand & constraint analysis and System Architecture will be illustrated in this chapter. The later design process and implementation will be introduced in following chapters.

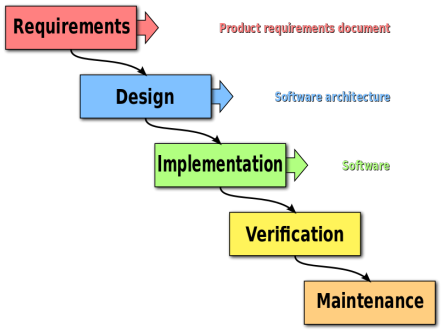

Figure 3.1: Diagram of waterfall model.(Adapted from Paul Smith's work at wikipedia)

3.1 Demand analysis

According to Dix, Alan[10], the most fundamental principle of web application design is to understand users' demand. Hence, the primary step of this project is to set system goals based on demand analysis. This section introduced essential functionalities for the proposed online trading platform. A concise description of each function will be presented as well.

In accordance to existed online trading platforms for Bitcoin such as Bitstamp, Bitfinex and Btctrade, the key elements and functions selected to implement in this project are listed as below:

1. Register/Login: In order to do transactions on this web platform, the Register/Login service is a necessary function to provide access for new users to register and old users to login.

2. Account Management: After login, through Account Management function, the system should enable users to change and update their previous inputted account information.

3. Deposit/Withdraw Money: Before doing the Bitcoin transactions, the platform should allow users to deposit money from other currency accounts (e.g. bank card, paypal) to its own currency account, and withdraw money from its own account to other accounts. This platform should be able to support multi-currency service, however the deposit and withdraw operations are only feasible for accounts with same currency.

4. Doing Transactions: In order to complete transactions according to users' requests, an accurate and efficient trading engine are proposed to be implemented.

The principle of Bitcoin trading is illustrated as below: when a "Buy transaction" was requested, the system will find matched "Sell transaction" (lower than bid price) with lowest asking price from the existed sell transaction requests in the platform to complete the transaction. Accordingly, when a "Sell transaction" was requested, the system will find the "Buy transaction" request with highest biding price from the existed Buy transactions in the platform to do the transaction. Furthermore, if there were more than one buy/sell transactions with the same lowest/highest price, the system will choose the earliest order to make a deal.

In addition, the order cancellation function should be developed to enable users to

cancel uncompleted order and partially-completed order.

An user-friendly trading interface should be designed for these core functions of the trading platform.

5. Operation Records Searching: Registered users should be able to check their currency account balance, transaction status and trading records. To do this job, the platform should be implemented with a records history searching function.

3.2 Constraints analysis

As introduced above, the initial goals of the online trading platform comprise five main functionalities, and it is important to take different kinds of constraints into account. The constraints for the implementation of this platform can be classified into following three categories.

1. Finance Principle Constraints: Currency transaction is a crucial part of a trading system. In order to deposit/withdraw money on the website application, cooperations with banks or some online payment companies like Paypal are required. In this project, a virtual payment system will be implemented to simulate currency transactions, instead of using real transaction system.

2. Computer Resource Constraints: Implementation of the online trading platform requires a specific server which is equipped with high performance processors and abundant memory. For easy to run and test, a laptop was planned to be the server of this project. Therefore, the hardware performance constraints should be considered when conduct the system performance evaluation.

3. Time Frames: The total scheduled time of this project is two months. A great amount of work are scheduled to complete within this time frame:

- Learning new programming languages and technologies, building developing environment;
- System architecture;
- Design front-end and back-end;
- Implementation of front-end and back-end;

- Basic testing, functionality testing and load testing;
- During the whole period of the project, documenting the process and writing report.

All these constraints need to be considered before the formal design work.

3.3 System Architecture

Once demand and constraint analysis were completed, the next step is to build an overall system architecture for this online trading platform. Classical software architecture pattern named Model-View-Controller (MVC) is applied in this project. The MVC pattern is constituted by three interconnected parts: the model, the view and the controller. As showed in Figure 3.2, each component in MVC is responsible for dealing with specific functions and can interact with each other conveniently. Since this design pattern can decouple the components of an application, it performs efficiently in code reusing and parallel development.

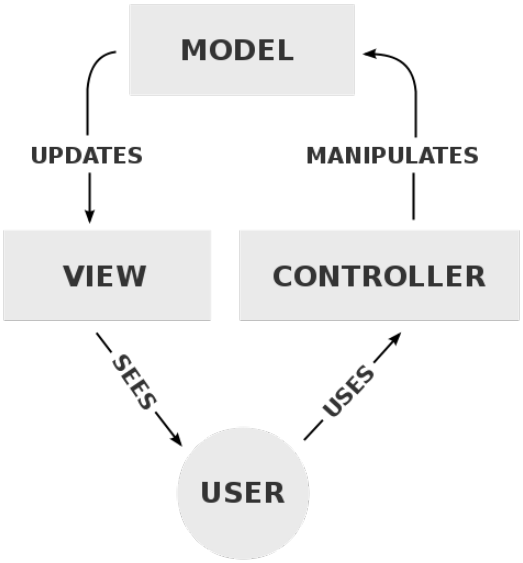

Figure 3.2: Diagram of interactions within the MVC pattern.(Adapted from https://en.wikipedia.org/wiki/Modelviewcontroller)

The Model component corresponds the data-related logic that users works with. It can interact with database directly, including manipulating and updating the data logic. The View component is used for UI logic of the web application, and is consisted of

web pages that allow users to interact with. Controller acts as an interface between Model and View components to deal with the business requests from website, which is capable of manipulating data through corresponding data logic Model and rendering the response to website again.

By using MVC, we propose to design the architecture with front-end and back-end. Specifically, the architecture structure of this online trading platform is illustrated as following Figure:

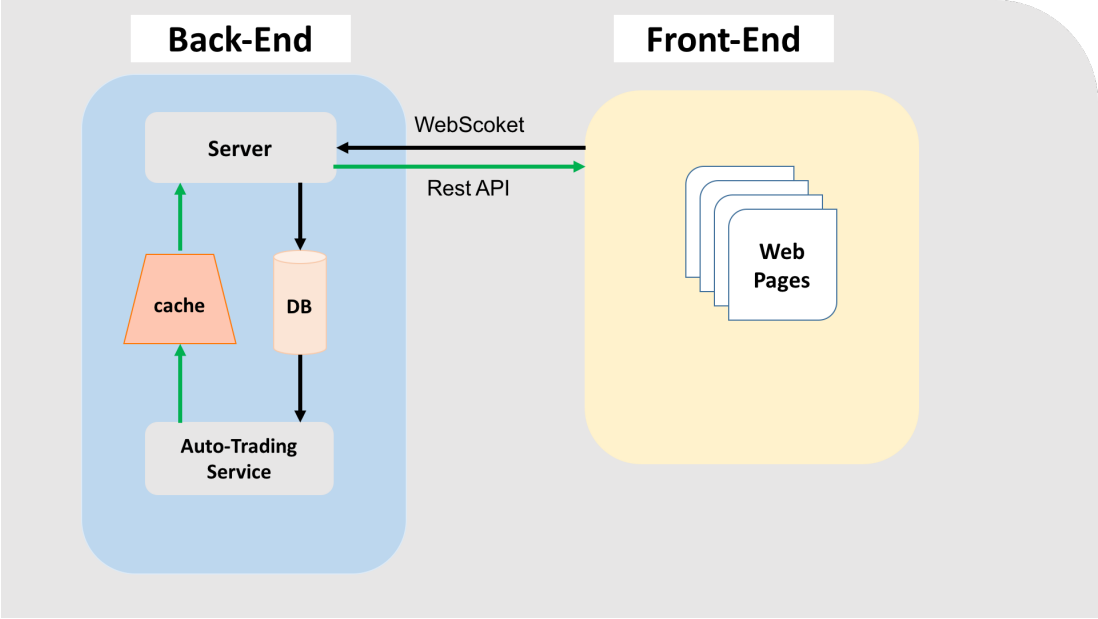

Figure 3.3: The Architecture of this trading platform

As showed in 3.3, the front-end and back-end are the main components of the architecture of the platform. Several web pages with different functionalities made up of the front-end. Each functionality corresponds a particular processing module in the back-end. Users can do a series of operations such as placing order, transferring money and account management on these web pages. The operation information will then be passed to back-end through web socket API. The Server module implemented in the back-end will process these received requests, complete information analysis, send related data to Auto-Trading engine, persist data-logic into Database (DB) and give response to web page to display at last.

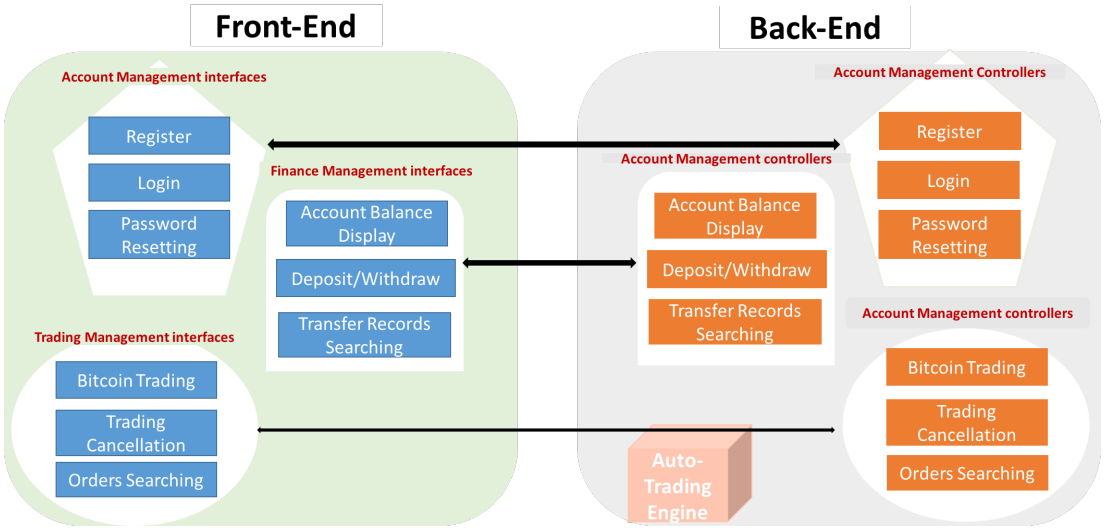

Figure 3.4: The functionalities of the front-end and back-end

The Auto-Trading engine in back-end acted as an auto-trading processor role in this system. The scheduled jobs were sent to this trading engine to do transactions during a certain time period. The details about Auto-Trading will be discussed in chapter 5 focused on ”Back-End Design”.

3.4 Summary

Through this pre-design stage, business demands of the proposed online trading platform have been analyzed thoroughly, which can provide preliminary design guidelines for later system design. While it is not possible to identify all system requirements just from theoretical analysis of the existed platforms, more complex situation encountered during the whole system design process will be carefully considered. The initial system architecture based on MVC pattern outlined the whole development processes, and clarified the logics between different components. In following chapters, the detail front-end and back-end design and implementation will be presented.

Chapter 4

Front-End Design and Implementation

According to software design principle[8], a typical developing iteration comprises following crucial steps: design, prototype testing, evaluation and optimized design. For web application development, multiple design iterations are needed to achieve acceptable performance. This online trading platform was deigned on the basis of the same iterative principle. In this chapter, the final designed front-end will be presented.

Since the usability is one of the key quality attributes for web application design[10], a series of user-friendly interfaces were designed for this online trading platform by HTML, Bootstrap and AngularJS. The front-end design will be introduced from four sub-sections, with each one covered a specific functionality interface design. The implementation process for each interface will also be illustrated in detail.

4.1 Account Management Interfaces

The account management module includes Register Interface, Login Interface and Password Resetting Interface. The design and implementation of each interface will be introduced separately for each sub-interface.

4.1.1 Register Interface

4.1.1.1 Functions and Elements

The Register Interface is one of the most basic interface for online trading platform. With registered accounts, users are able to save their corresponded information and submit requests on this website application. In addition, these accounts can help system to remember each customer's behavior and distinguish them from other customers.

In this project, email and password were decided to be the main elements of Register Interface. With valid email accounts, users can register accounts on this system conveniently. Once successfully registered on the platform, users can login with their email and password and then have access to trading services which are only provided to registered users. Additionally, the interface was designed to support basic JQuery validation.

4.1.1.2 Implementation

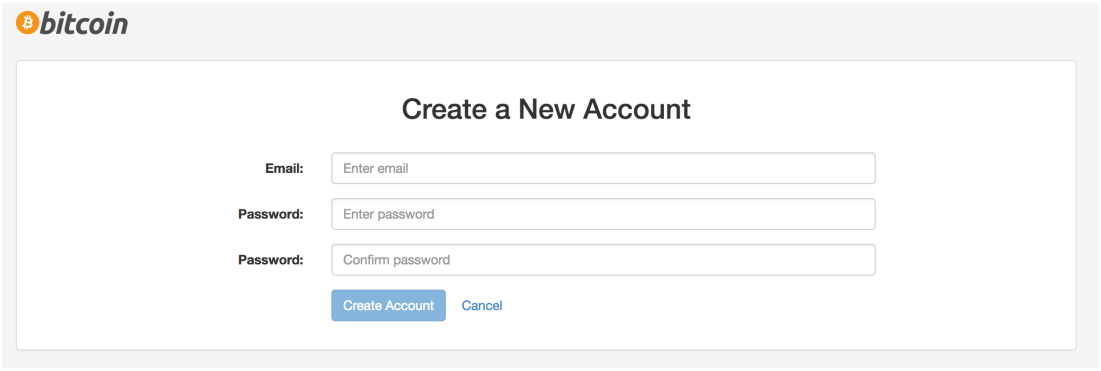

Figure 4.1: The display of Register interface

1. **Interface Display:** As the Figure 4.1 showed, the Register Interface contained a register form which was consisted of "email" input box, "password" input box, "confirm password" input box, "create account" button and "cancel" button.
2. **Interface Validations:** This interface was implemented to support JavaScript Form Validations. The validation tasks were:
    - Has the user filled in all required fields: Email, Password and Confirm Password?

- Has the user provided a valid email?
- Has the user filled the same value in Password and Confirm Password?

The ”Create Account” button will be enabled (highlighted) only if all above validations were passed. Otherwise, a corresponded error message will be alerted when users violated any of these validations(4.2).

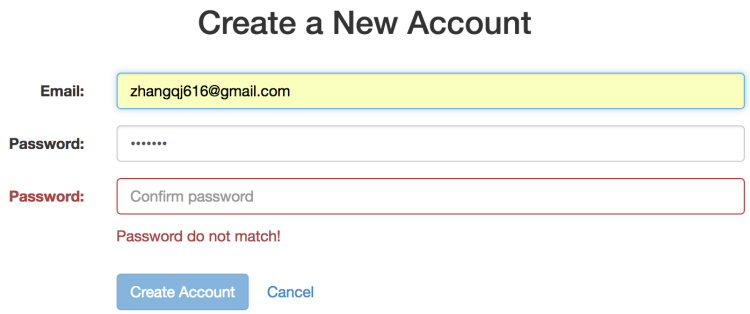

Figure 4.2: The validation of Register Interface

3. **Interface Actions:** When users click ”Create Account” button, the data filled in the register form will be packaged with Json format and sent to back-end corresponding Register Controller to process by using HTTP Request. If back-end responded registering successfully, it will render to Login Interface, otherwise, error messages will be displayed on the register page to notify users. If uses clicked the ”Cancel” button, it will render to Login Interface directly.

4.1.2 Login Interface

4.1.2.1 Functions and Elements

After successful registration, users can use the registered email and password to login the platform. This platform was developed to be able to keep the login status when exploring among different sub pages on this website unless the log out operation was activated.

4.1.2.2 Implementation

1. **Interface Display:** As the Figure 4.3 showed, the Login Interface contained a

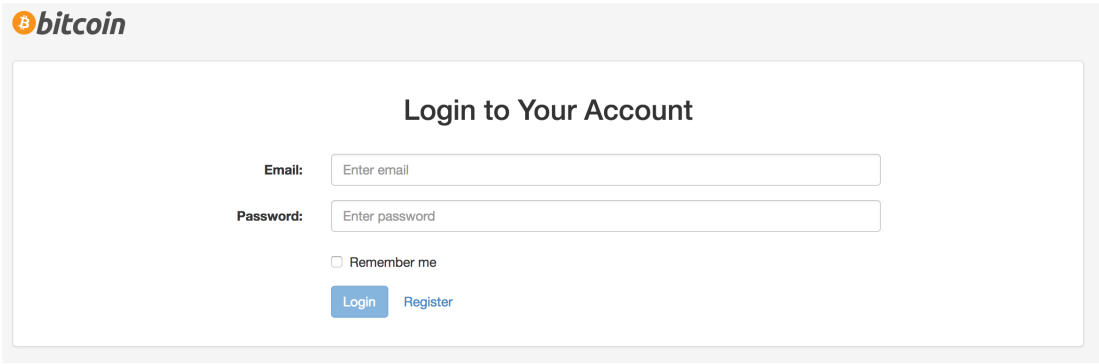

Figure 4.3: The display of Login Interface

login form which was consisted of ”email” input box, ”password” input box, ”Login” button and ”Register” button.

2. **Interface Validations:** This interface was implemented to support JavaScript Form Validations. The validation tasks were:

   - Has the user filled in all required fields: Email, Password?
   - Has the user provided a valid email?

   The ”Login” button can be enabled only if all above validations were passed. Notification messages will appeared as Figure 4.4 showed if users violated any validations.

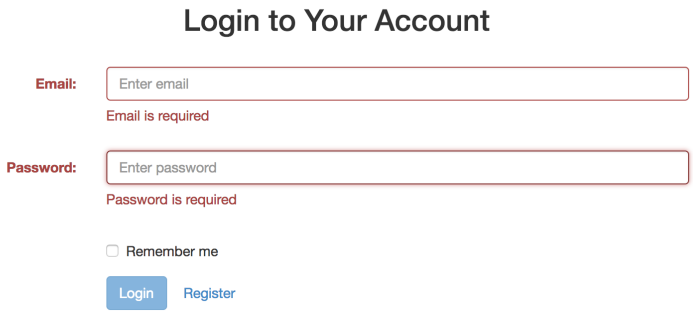

Figure 4.4: The validation of Login Interface

3. **Interface Actions:** When users click ”Login” button, the data filled in the login form will be packaged with Json format, and sent to back-end corresponding Login Controller to process by using HTTP Request. If the email and password were correct, it will render to Home Page (Figure 4.7) after the process of back-end, if not, error message will be showed. If users clicked the ”Register” button,

it will render to Register Interface which was presented above.

4.1.3 Password Resetting

4.1.3.1 Functions and Elements

After successful login to the platform, users can update their password in Password Resetting Interface. This interface was designed to contain three main elements, including user account, old password and new password.

4.1.3.2 Implementation

1. **Interface Display:** As the Figure 4.5 showed, the Password Resetting Interface

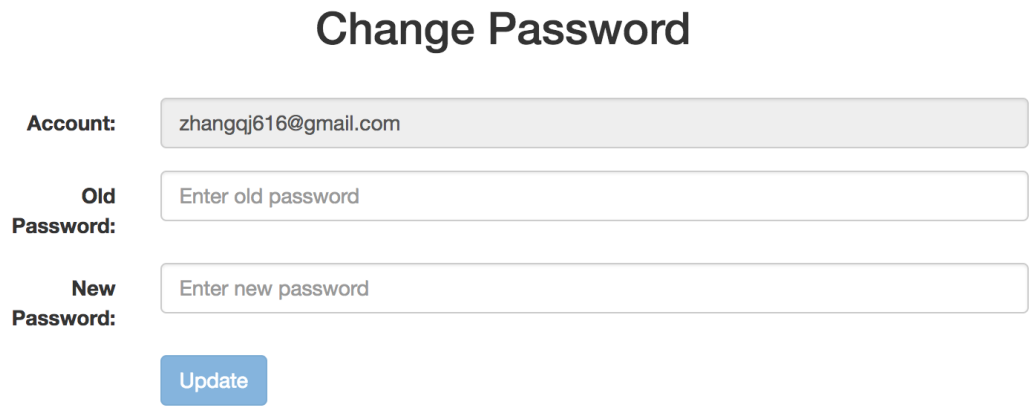

Figure 4.5: The display of Password Resetting Interface

contained a resetting form which was consisted of ”Account” input box, ”Old Password” input box, ”New Password” input box and ”Update” button. The ”Account” input box was read-only and the value was filled automatically when the interface was opened.

2. **Interface Validations:** This interface was implemented to support JavaScript Form Validations. The validation tasks were:

   - Has the user filled in all required fields: Old Password, New Password?

   The ”Update” button can be enabled only if all above validations were passed. In addition, the error messages will be alerted when users violated these validations(4.6).

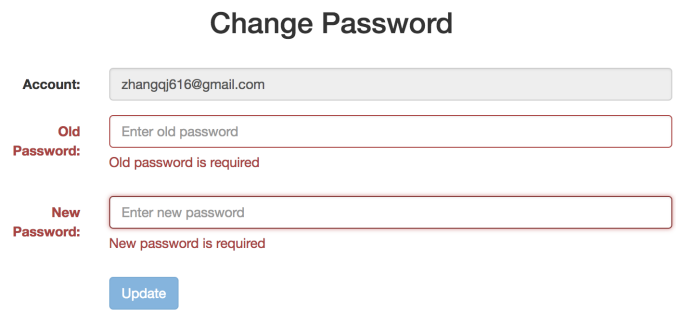

Figure 4.6: The validation of Password Resetting Interface

3. **Interface Actions:** When users clicking ”Update” button, the data filled in the update form will be packaged with Json format, and sent to back-end corresponding Password Update Controller to process by using HTTP Request. It will render to the Home page(Figure 4.7) if back-end gave successful update feedback,otherwise, it will show the error message to users.

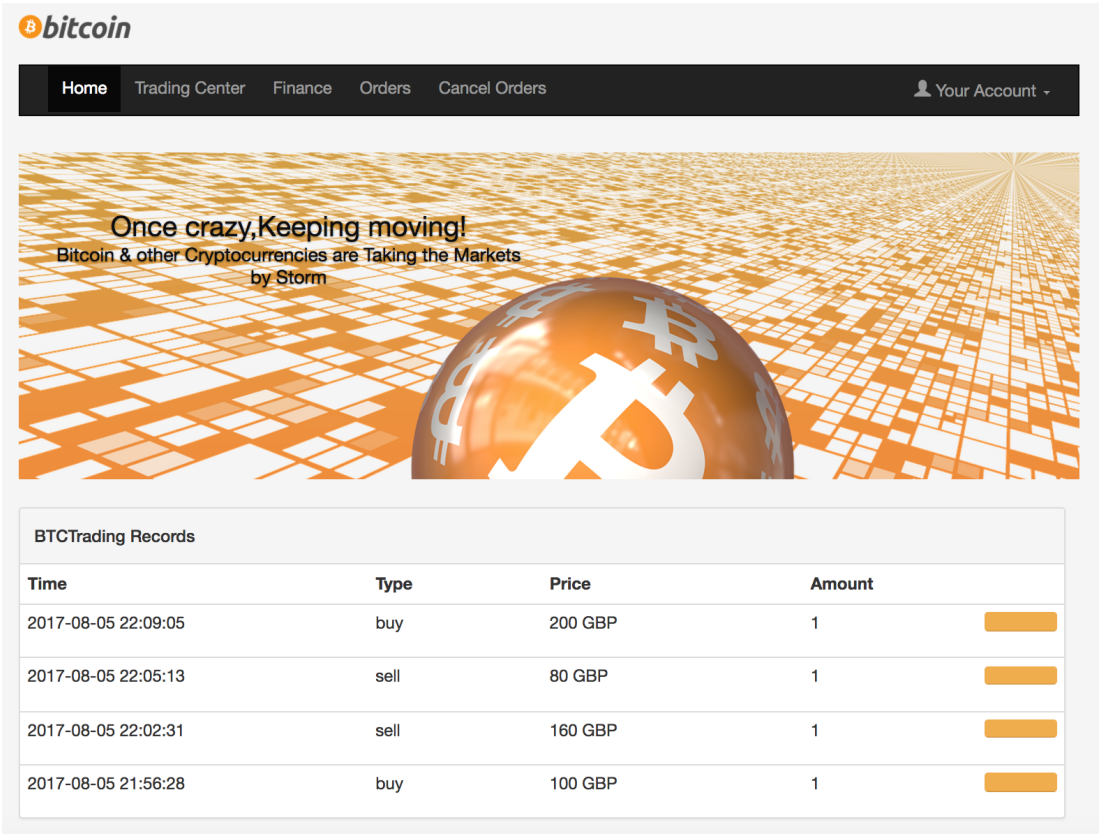

Figure 4.7: The display of trading home page interface

4.2 Finance Management Interfaces

Finance Management is the essential part of trading platforms, which allow users to manage their account balance. According to traditional finance service, this platform was determined to provide three typical finance services:

1. **Account balance display**.
2. **GBP/CNY/Bitcoin currency deposit/withdraw service**
3. **Account Transfer records searching service**.

For the currency deposit and withdraw service, this platform provided virtual transactions environment to simulate the real trading business between system account and other currency accounts like bank cards and paypal accounts. British pound, Chinese Yuan and Bitcoin are selected and implemented in the system as different panels.

In this section, three interfaces will be described in detail, including Account Balance Interface, GBP Deposit/Withdraw Interface, Transfer Records Searching Interface. Although there were three different interfaces for three different types of currency, the functions and elements of each interface were identical. Therefore, only one typical currency deposit/withdraw interface will be introduced in this chapter.

4.2.1 Account Balance Interface

4.2.1.1 Functions and Elements

The Account Balance Interface was designed to display the account balance of different types of currency. The content should include currency type, available balance, frozen balance and total balance.

4.2.1.2 Implementation

1. **Interface Display:**The figure 4.8 displays the overview of Account Balance Interface. Three different types of currency balance were displayed in this interface.

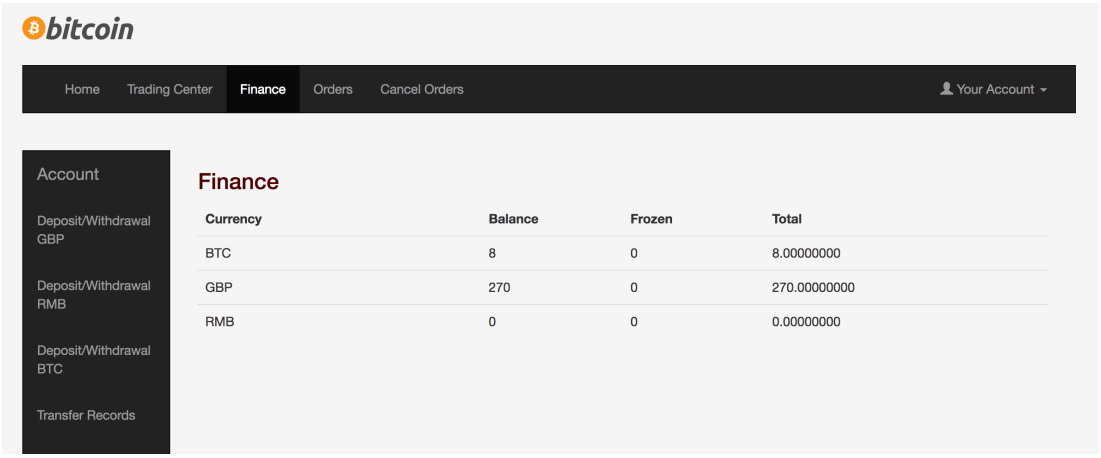

Figure 4.8: The display of Account Balance interface

2. **Interface Validations:** Since this interface is used to display balance and has no input fields, there was no interface validation here.

3. **Interface Actions:** User's account balance will be display as soon as the interface is opened. In order to get the balance of user account, an action was developed in the JavaScript initial function of the Account Balance Interface to send request to Finance Management Controller in the back-end.

4.2.2 GBP Deposit/Withdraw Interface

4.2.2.1 Functions and Elements

This interface was designed to transfer money between system account and other finance accounts such as bank cards. Therefore, transfer amount, account number and transfer password should be included in the GBP Deposit/Withdraw Interface.

4.2.2.2 Implementation

1. **Interface Display:** As the Figure 4.9 showed, the "balance" field presented the current money balance for selected currency. After users inputted a card number, a certain amount of money and correct password in the deposit or withdraw panel, the deposit/withdraw button will turn from disabled to enabled.

2. **Interface Validations:** This interface was implemented to support JavaScript Form Validations. The validation tasks were:

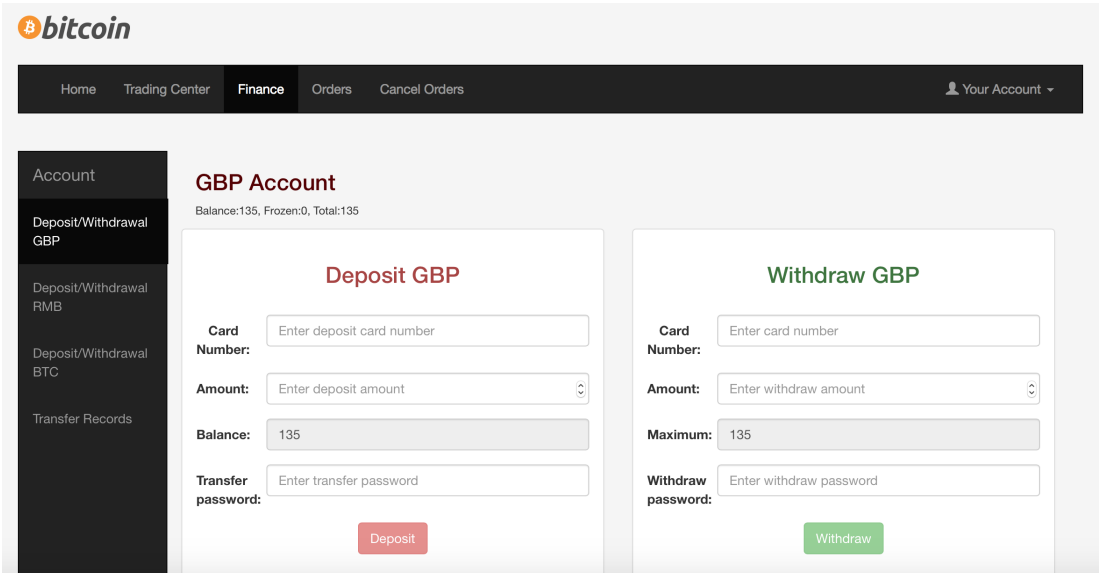

Figure 4.9: The display of GBP Deposit/Withdraw interface

- Has the user filled in all required fields: Card Number, Amount and Transfer Password?
- Has the user entered number format amount?
- Has the user withdraw amount greater than the maximum withdraw amount?

The deposit/withdraw buttons kept disabled until the users filled all necessary input and passed all validations. Again, the error messages will be alerted when users violated these validations(4.10).

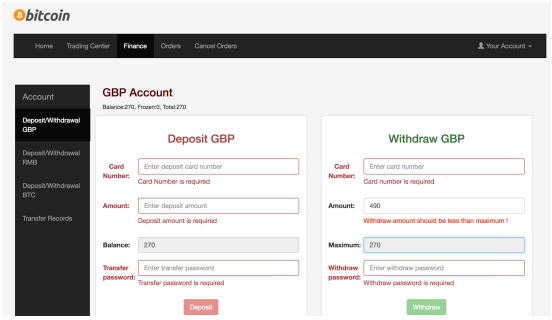

Figure 4.10: The validation of GBP Deposit/Withdraw interface

3. **Interface Actions:** By clicking deposit/withdraw button, the data filled in the deposit/withdraws form is packaged with Json format, and sent to back-end corresponding Deposit/Withdraw Controller to process by using HTTP Request. When the transfer transaction was completed, the exact amount of money which

user deposited or withdrew will be added or subtracted from the system account balance accordingly.

4.2.3 Transfer Records Searching Interface

4.2.3.1 Functions and Elements

For the transfer records history searching service, a records searching interface was designed to support customers to check history transaction records on the platform by selecting a certain time span. The records searching interface will display all history records which happened during that period.

4.2.3.2 Implementation

1. **Interface Display:** As the Figure 4.11 showed, the ”Period” field was repre-

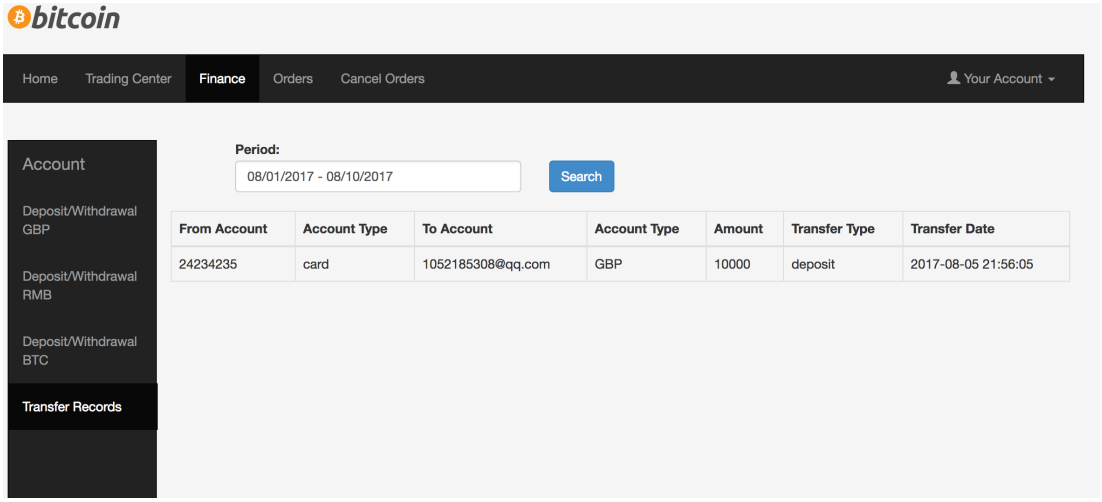

Figure 4.11: The display of Transfer Records Searching interface

sented the searching time span. The records list below searching field showed transfer history records which matched the searching condition. Each record list with relevant basic transfer information, including From account, To account, Transfer amount and Transfer time.

2. **Interface Validations:** Since this interface is used to display transfer history record and the only input box is an bootstrap-datepicker, there was no interface validation here.

3. **Interface Actions:** By clicking Search button, the value of ”Period” field is packaged with Json format, and sent to back-end corresponding Transfer Searching Controller to get the satisfied transfer records by using HTTP Request. When the searching operation was completed, the transfer records will be displayed in the records list.

4.3 Trading Management Interfaces

Doing Bitcoin trading on the website is the key motivation for this proposed online trading platform. In light of this, the trading interface design is crucial for this platform. The trading section on this platform was equipped with three trading interfaces: Bitcoin Trading Interface, Trading Cancellation Interface and Orders Searching Interface.

4.3.1 Bitcoin Trading Interface

4.3.1.1 Functions and Elements

This proposed BitCoin online trading platform applied same classical trading pattern used in other existed trading platforms. For classical trading pattern, trading requests are placed based on inputting bid/ask price and amount of Bitcoin. In addition, customer should be able to choose different currency types for placing orders. To avoid trading error, the system need to pre-check that sufficient money or Bitcoin are available for relevant buying request or selling request. Otherwise, the request cannot be submitted successfully, and error notification should be designed to inform customers to modify the order information. Based on the design requirements which were illustrated in above Functions and Elements section, the Bitcoin trading interface was developed with above essential elements and functionalities as showed in 4.12.

4.3.1.2 Implementation

1. **Interface Display:** As the above figure illustrated, the trading interface contains two independent sections. The above section is consisted of BuyBTC panel and SellBTC panel, which can be used to submit buy and sell bitcoin requests. The

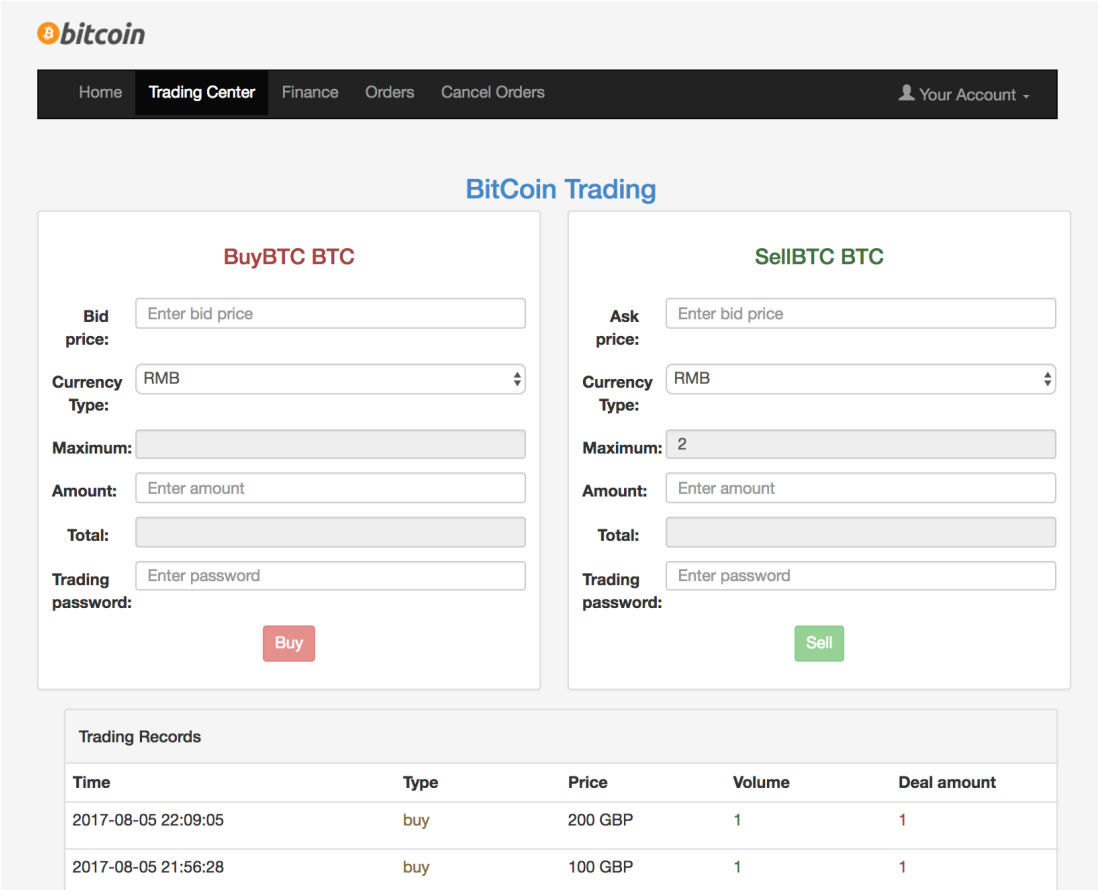

Figure 4.12: The display of Bitcoin Trading interface

below section includes a Trading Records panel, that is used to display user's trading records. For each trading panel, required input are bid price/ask price, amount of Bitcoin and currency type. The "total" value represent the product of bid/ask price and amount of Bitcoin, which will be calculated automatically from input data. The "maximum" in BuyBTC panel will display the maximum amount of Bitcoin the current customer is able to buy based on the bid price and available account balance. In SellBTC panel, the "maximum" filed is the exact balance of current customer's BTC account, which is also the maximum amount of bitcoin available to sell.

In this trading interface, all trading requests placed by the customer will be recorded and showed in the Trading Records panel. The history orders panel was designed to perform like a dashboard which can display information and refresh automatically.

2. **Interface Validations:** This interface was implemented to support JavaScript

Form Validations. The validation tasks were:

- Has the user filled in all required fields: Bid Price/Ask Price, Currency Type, Amount, Trading Password?
- Has the user entered number formated amount and price input box?
- Is there sufficient money for Buying request? or is there sufficient Bitcoin for Selling Request?

The Buy button and Sell button will be activated (highlighted) until all fields in BuyBTC panel or SellBTC panel have been filled correctly and passed all validations, which will guarantee that all submitted requests are valid.

3. **Interface Actions:** By clicking deposit/withdraw button, the data filled in the deposit/withdraw panel will be sent to Trading Processing Controller in back-end with Json pattern to handle by using HTTP API. When the trading requests were processed successfully, the trading records will be displayed in the Trading Record Panel, otherwise error message will be showed to users.

4.3.2 Trading Cancellation Interface

4.3.2.1 Functions and Elements

Trading cancellation is a necessary function for a mature trading platform. This platform was developed to support cancellation of uncompleted order and partially completed order. The cancellation history will be displayed along with order status.

4.3.2.2 Implementation

1. **Interface Display:** As showed in Figure 4.13, the cancellation interface is constituted of three subsections. The above panel named ”BTC Trading Record History” displays all past orders with a progress bar which represents the order status. A ”cancel” button is added for each order to submit order cancellation request. Additionally, the following two subsections presents successfully canceled orders and failed canceled orders separately. The records information in these three panels will refresh frequently to let the customer know the orders progress rate in real time.

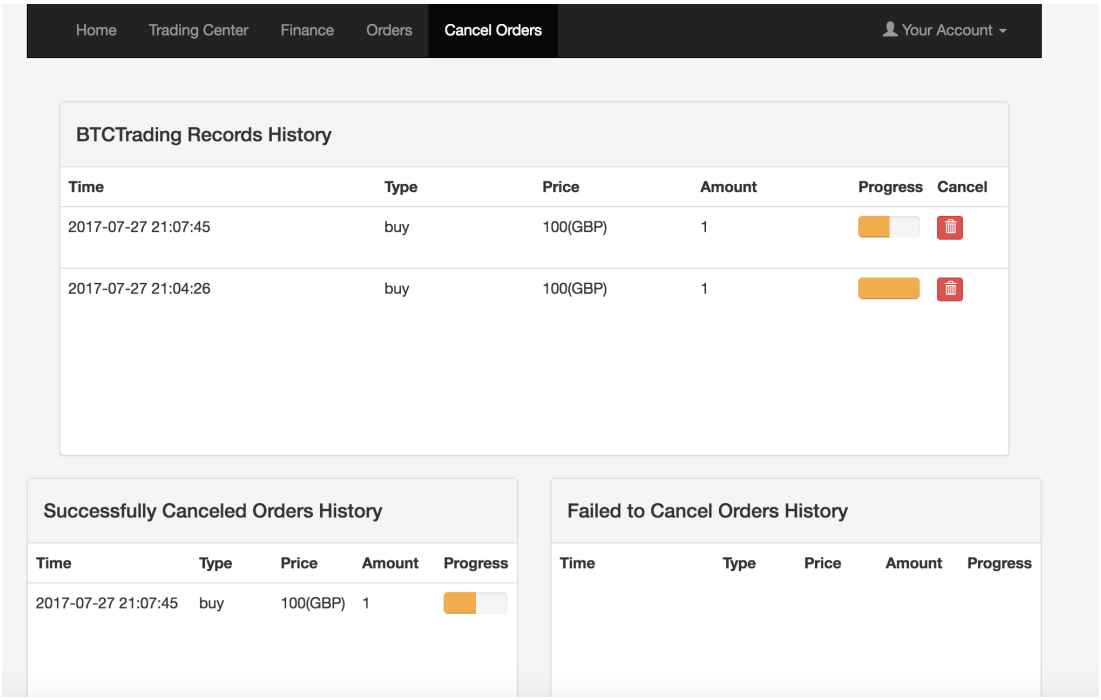

Figure 4.13: The display of Trading Cancellation interface

2. **Interface Validations:** Since there is no input box in this interface, no data validation applied here. However, a double-check cancellation dialogue box (shows in 4.14) was implemented to avoid that users make cancellation accidentally.

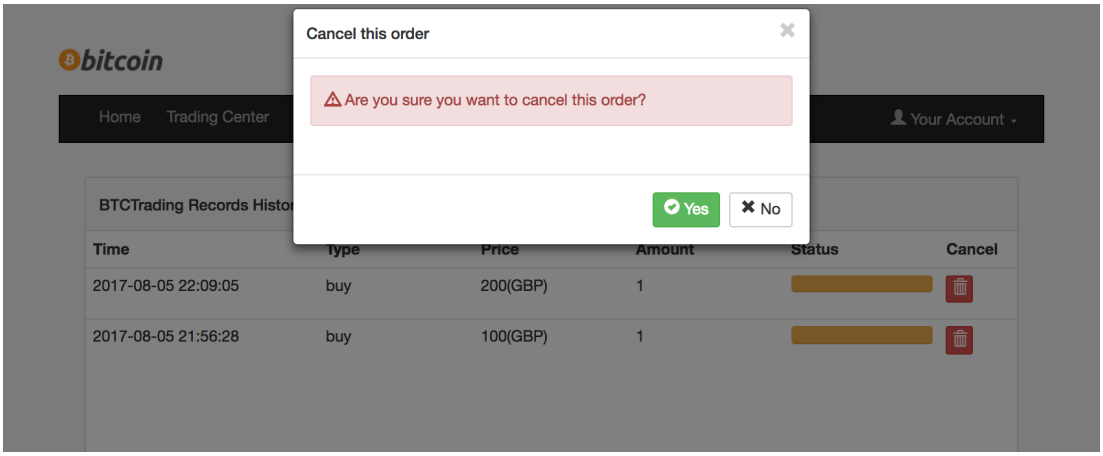

Figure 4.14: The display of Cancellation double-check box interface

3. **Interface Actions:** By clicking the ”yes” button in the double-check box showed in 4.14, the trading cancellation request will be sent to Cancellation Controller in back-end to process. When the cancellation was completed successfully, the cancellation records will be displayed in the successfully canceled orders panel,

otherwise, the failed cancellation operation will be showed in the failed canceled orders panel.

4.3.3 Orders Searching Interface

4.3.3.1 Functions and Elements

Once the trading requests were completed, the corresponding trading information will be presented in ”Orders” panel with final trading price. A time searching function is implemented to present history trading orders within certain time span.

4.3.3.2 Implementation

1. **Interface Display:** As the Figure 4.15 showed, the ”Period” field was repre-

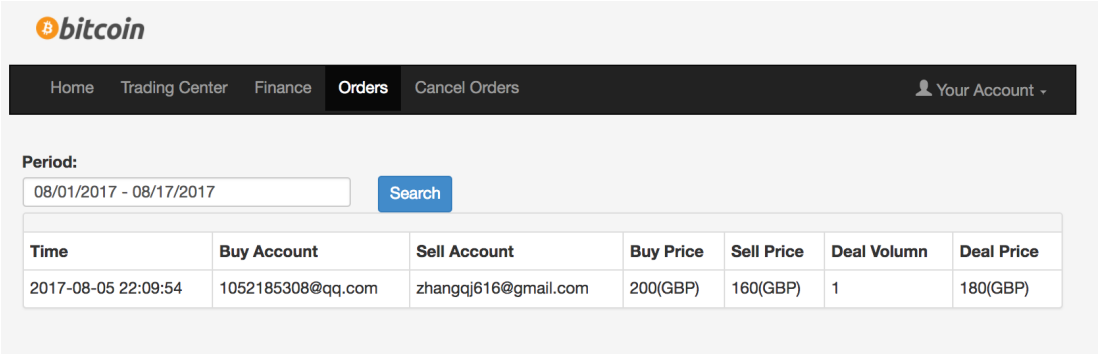

Figure 4.15: The display of Orders Searching interface

sented the searching time period. The records list below showed the trading history records which matched the searching condition. Each record in the list was consisted of basic trading information, including Time, Buy account, Sell account, Buy price, Sell price, Deal price and Deal amount.

2. **Interface Validations:** Since this interface is used to display trading order history records, and the only input box is an bootstrap-datepicker, there was no interface validation here.

3. **Interface Actions:** By clicking Search button, the value of ”Period” field is packaged with Json format, and sent to back-end corresponding Trading Order Searching Controller to get the satisfied traded records by using HTTP Request.

When the searching operation was completed, the traded records will be displayed in the records list.

4.4 Summary

From above front-end design, all required interfaces for this online trading platform have been successfully implemented. A series of validations were integrated through AngularJS to improve user experience and avoid system errors. The detail implementation of functions provided by these front-end interfaces will be completed in the back-end design, and will be described in chapter 5.

Chapter 5

Back-End Design and Implementation

The back-end is the most fundamental part of this trading platform which enables front-end to provide relevant services. Several novel technologies have been applied in the back-end design, including Scala, MongoDB and Play framework. Given technology challenges and huge workload, the majority of time and effort were devoted to this part. The detail design process and internal data logic will be illustrated by following three sub-sections:

- Date Schema Design;
- Basic Functionalities Design;
- Auto-trading Engine Design.

5.1 Data Schema Design

Data schema design has significant influence on the effectiveness and efficiency of application systems. Based on the Database Design Flow Diagram developed by Navathe and Schkolnick's [18], data schema design can be divided into four main steps:

1. **Requirement Analysis**
2. **Conceptual Schema Design**
3. **Implementation Design**
4. **Physical Schema Design and Optimization**.

The same design process is applied in this project. The only difference is that since MongoDB is selected to be the system database and it is a no relational database, therefore, the fourth step physical schema design can be ignored. The first three design steps are introduced in detail in following sections.

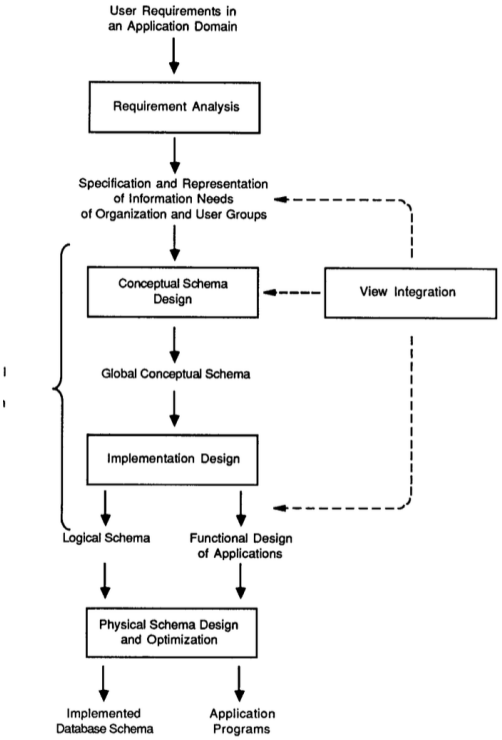

Figure 5.1: Phases of database design. (Adapted from Navathe and Schkolnick [1978].)

5.1.1 Requirement Analysis

According to the demand analysis in Project Overview section, a number of business goals including register/login, password resetting, account transfer, doing Bitcoin trading and records history searching were supposed to be implemented in this platform. In light of this, the platform data schema was designed to include following four main components:

1. Account Models: To realize the functions in Account Management Interfaces, a set of account models should be created to manipulate data from register interface, login interface and password resetting interface.

2. Finance Models: In order to implement the functions in Finance Management Interfaces, a set of finance models should be built to save transfer records, update

account balance and connect with database.

3. Trading Models: A set of trading models should be created to operate trading data, such as saving trading records and updating trading amount.

4. Task Models: A trade task model and a trade task flow were supposed to store trading request jobs which will be processed in the auto-trading engine.

5.1.2 Conceptual Schema Design

On the basis of requirements analysis presented in the above section, a number of logical data models of the business information were built up. The following figure shows the overview of the data model structure of this system.

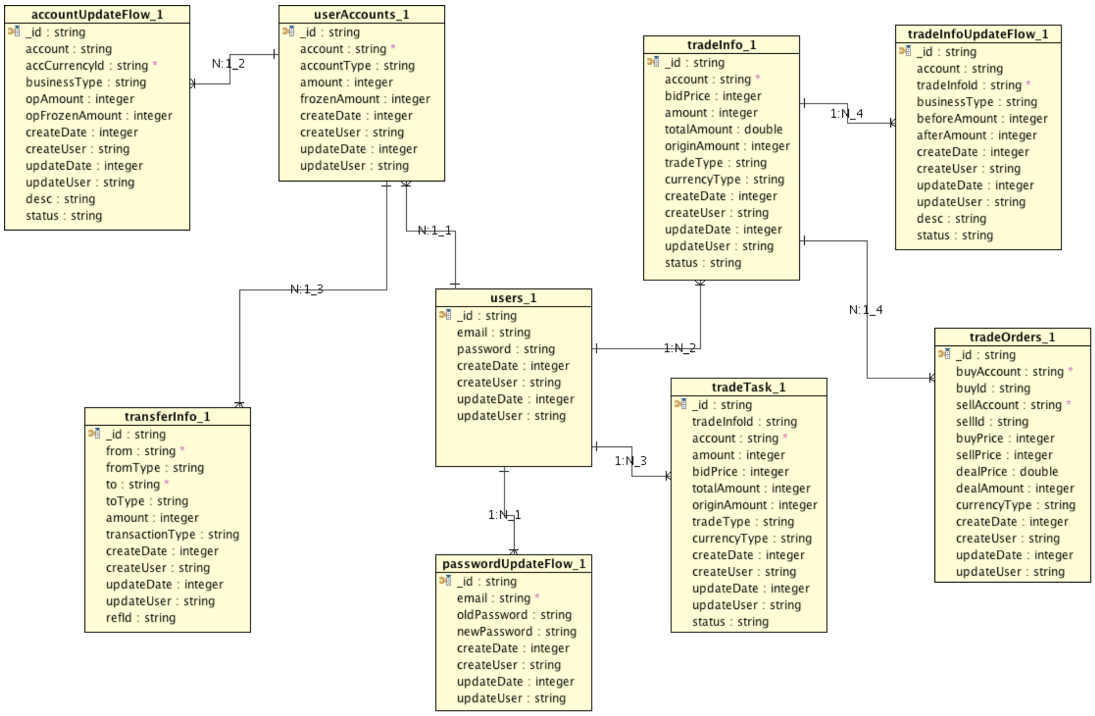

Figure 5.2: The data model design of the platform

As the above data schema figure showed, the system business requirements were transformed into nine data schema. Each schema consists of several elements to represent specific attributes of business entities. Every data model contain the same four attributes: createDate, createUser, updateDate, updateUser. These four attributes were used to document operation information which can help to improve the security of system. Furthermore, four flow data schema were designed to record modification history

of the corresponding main schema. For instance, the ”passwordUpdateFlow” schema was created to store the modification history of the user password. One of the advantages of this deign approach is to make the modification of important information can be retrieved.

5.1.3 Implementation Design

In order to implement the proposed data schema in this platform, a model class was created for each data model. In each model class, a collection with corresponding table in database was defined to allow applications to operate database . A typical implementation of this kind of schema is displayed as the following codes:

```
case class TradeInfo (
                  id:Option[String],
                  account: String,
                  bidPrice : BigDecimal,
                  amount : BigDecimal,
                  totalAmount:BigDecimal,
                  originAmount:BigDecimal,
                  tradeType :String,
                  currencyType: String,
                  createDate:Option[DateTime],
                  createUser: String,
                  updateDate:Option[DateTime],
                  updateUser:String,
                  status:Option[String]
                )

object TradeInfo{
  lazy val reactiveMongoApi =
  current.injector.instanceOf[ReactiveMongoApi]
  val collection = reactiveMongoApi.db.collection
  [JSONCollection]("tradeInfo")
}
```

5.2 Basic Functionalities Design

Three main basic services as showed in front-end are introduced in following sections. Each of these service was implemented by a corresponding Controller which is used to process requests from front-end interfaces and manipulate data using the pre-defined data model in Data Schema Design stage.

5.2.1 Account Management Service

The account management services are consisted of Register Service, Login Service and Password Resetting Service. A number of APIs were provided to handle requests from each service, and each API requires several parameters to complete corresponding functional processing:

1. Register Service
    - **API:** /accountApp/create/
    - **Parameters:** user { email, password }
    - **Function:** Create user if the email address has not registered.
2. Login Service
    - **API:** /accountApp/authenticate/
    - **Parameters:** user { email, password }
    - **Function:**Authenticate the user to login if the email and password are correct.
3. Password Resetting Service
    - **API:** /accountApp/update/
    - **Parameters:** user{ email, oldPassword, newPassword }
    - **Function:**Update password if the condition is satisfied.

5.2.1.1 Register Service

For register service, the register information packaged with Json format was sent to the Register Controller from the Register Interface in front-end. When the Register Controller received these information, following several steps will be processed:

1. Query user information from **"Users"** table by using "email" from register request.
2. If the user with same email was existed, render **"Email is already registered!"** message to front-end to notify customers the invalid reason.
3. If not existed, encrypt the password by using BCrypt, then store the user information into "Users" table in MongoDB, meanwhile three initial currency balance records with amount equals **0** are inserted into **"userAccounts"** table. If store successfully, return **"Registered successfully !"** message to front-end, if not, give **"Registration failed !"** message.

5.2.1.2 Login Service

For login service, the login information was packaged with Json pattern and sent to Login Controller in the back-end. When the Login Controller received this request, it will do following steps:

1. Query user information from **"Users"** table by using "email" from login request.
2. If the user was not existed, give **"Invalid email !"** message to front-end to notify the login failure reason.
3. If the user was existed, compare the requested password with the password stored in database. If the password was same, given **"Login successfully"** message to users and render to Trading Home page, If not, respond **"Invalid password"** to front-end.

5.2.1.3 Password Resetting Service

For password resetting service, a Password Resetting Controller was designed in back-end deal with resetting request from the front-end. The processing logic can be described as following steps: As Figure5.3 showed, There are four steps to process pass-

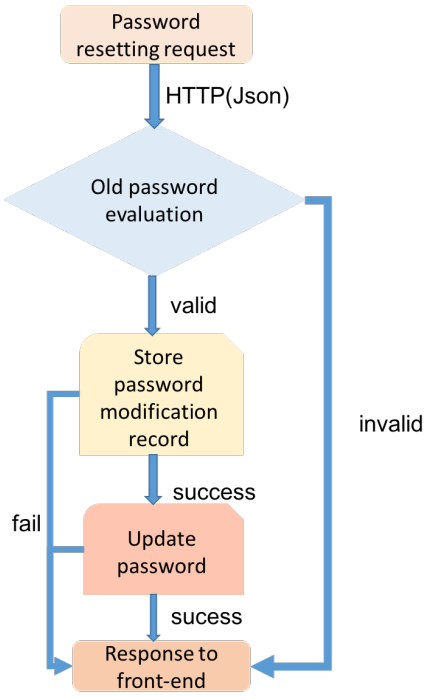

Figure 5.3: The phases of password resetting processing.

word resetting requests.

1. Query user information from **"Users"** table by using "email" from password resetting request.
2. Compare the requested oldPassword with stored password, if they are not same, return **"Old password is not matched !"** to notify users error reason. Otherwise, go to the next step.
3. Store password resetting flow into **"passwordUpdateFlow"** table, if stored successfully, turn to the 4 step. Otherwise, give error message to front-end.
4. Update user's password field with "new password" to **"User"** table, if success, give **"Update successfully"** message to front-end, if not, return **"Update failed"**.

5.2.2 Finance Management Service

The finance management service consisted of Account Balance Display Service, Deposit/Withdraw Service and Transfer Records Searching Service. A number of APIs were provided to handle requests from each service, and each API requires several

parameters to complete corresponding functional processing:

1. Account Balance Display Service
    - **API:** /financeApp/getAccountCurrency/
    - **Parameters:** userInfo { email }
    - **Function:** Get all currency account information of the user.
2. Deposit/Withdraw Service
    - **API:** /financeApp/DoTransfer/
    - **Parameters:** transferInfo { from, fromType, to ,toType,amount,refId,transferType}
      from: *transfer from account number*
      from type: *the type of transfer from account*
      to: *transfer to account number*
      to type: *the type of transfer to account*
      amount: *transfer amount*
      refId: *the record id of operated currency account.*
      transferType: *the type of transfer, withdraw or deposit*
    - **Function:** Do deposit or withdraw operation.
3. Transfer Records Searching Service
    - **API:** /financeApp/getTransferRecords/
    - **Parameters:** searchInfo{ email(user)}
    - **Function:**Search all transfer records of the user.
    - **API:** /financeApp/getTransferRecordsByDate/
    - **Parameters:** searchInfo{ email(user), startDate, endDate }
    - **Function:** Search all transfer records of the user which occurred time between startDate and endDate

5.2.2.1 Account Balance Display Service

For account balance display service, a Finance Management Controller was designed in back-end deal with account balance display request from the front-end. The processing logic can be described as following steps:

1. Query user account balance information from **userAccounts** table by using user's email.
2. Return the results list to front-end to display.

5.2.2.2 Deposit/Withdraw Service

In order to realize the multi-currency requirement, the customer account in this platform was designed to support three type of currencies including RMB, GBP and BTC. Therefore, the deposit/withdraw service support deposit/withdraw operation for all these three kinds of currencies.

The deposit/withdraw information packaged with Json format was sent to the Deposit/Withdraw Controller from the currency deposit/withdraw interfaces in front-end. When the Deposit/Withdraw Controller received these information, following several steps will be processed:

1. For withdraw request, check that the amount of currency to withdraw should not be greater than its current account balance. For deposit request, go to step 2.
2. Store deposit/withdraw information into **transferInfo** table. If store successfully, go to step3, or return error message to front-end.
3. Update the account balance with the same currency type in request, return successful message when update successfully and failed message when update with errors.

The processing of deposit/withdraw requests can be summarized in the following flow diagram Figure5.4:

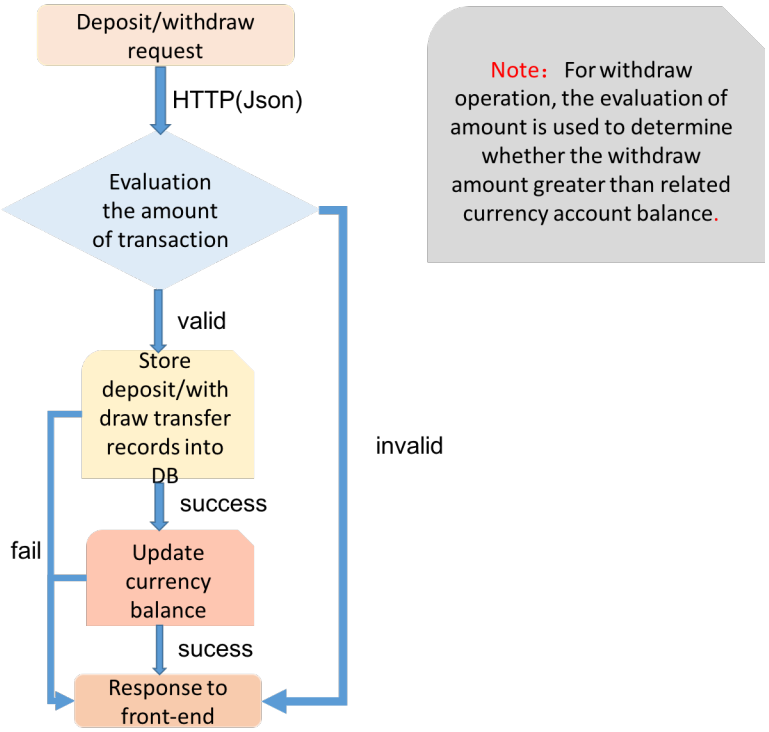

Figure 5.4: The phases of process deposit/withdraw processing.

5.2.2.3 Transfer Records Searching Service

For transfer records searching service, a Transfer Searching Controller was designed in back-end deal with transfer records searching requests from the front-end. The processing logic can be described as following steps:

1. Query transfer records from **transferInfo** table by using time period and the requested user's email.
2. Return the results list to front-end to display.

5.2.3 Trading Management Service

The trading management service can be divided into three components: Place Order Service, Order Cancellation Service and Order Searching Service. A number of APIs were provided to handle requests from each service, and each API requires several parameters to complete corresponding functional processing:

1. Place Order Service

- **API:** /tradeApp/DoTrading/
- **Parameters:** tradeInfo { account,amount,price,totalAmount,tradeType,currencyType }
  account: *the email of operating user*
  amount: *the trading volume of Bitcoin*
  price: *the price of buy or sell*
  totalAmount: *the total trading amount*
  tradeType: *the trading type, buy or sell*
  currencyType: *the trading currency type*
- **Function:** Do buy or sell trading

2. Order Cancellation Service
   - **API:** /tradeApp/doCustomerTradeOrderCancel/
   - **Parameters:** cancellationInfo { orderId}
   - **Function:** Do order cancellation operation.
3. Order Searching Service
   - **API:** /tradeApp/getCustomerTradingRecords/
   - **Parameters:** searchInfo{ email(user)}
   - **Function:**Search all trading records of the user.
   - **API:** /tradeApp/getCustomerTradingRecordsByDate/
   - **Parameters:** searchInfo{ email(user), startDate, endDate }
   - **Function:** Search all trading records of the user which occurred time between startDate and endDate.
   - **API:** /tradeApp/getCustomerTradeOrders/
   - **Parameters:** searchInfo{ email(user)}
   - **Function:** Search all completed trading orders of the user.

- **API:** /tradeApp/getCustomerTradingRecordsByStatus/
- **Parameters:** searchInfo{ email(user),status}
- **Function:** Search all trading orders of the user with a specific status.

5.2.3.1 Place Order Service

In order to handle high concurrency requests efficiently and give customer feedback quickly, the whole processing of buy/sell transactions was separated into two stages. The request will be first delivered into Trading Processing Controller and give instant feedback to customers with successfully submitted order information listed in Trading Records panel. Then these trading tasks will be allocated to the auto-trading engine to complete tasks, as this process is complicated and time consuming. The auto-trading engine design will be explained later in the ”Auto-trading Engine” section.

Once Trading Processing Controller received buy/sell requests from the front-end, a series of actions will be processed as Figure 5.5 shows.

1. Query account balance from **userAccounts** table by using currency type and account in the trading request.
2. Check whether the account balance is greater or equal than the trading amount, if not, return ”Account balance is not sufficient” message to front-end.
3. Store trading information into **”tradeInfo”** table with status = ”000” (”000” present initial state), meanwhile, store a trading task into **”tradeTask”** table with status = ”000”. If stored successfully, go to next step, otherwise, return error message to front-end.
4. Update **”userAccounts”** table by removing the request amount from ”amount” field to ”frozenAmount” field. If updated successfully, return **”Do Bitcoin trading successfully !”** to front-end, or return **”Do trading error !”**.

5.2.3.2 Order Cancellation Service

Similar to Place Order Service, the order cancellation tasks from Order Cancellation Service will be processed with two steps. The Order Cancellation Controller in the back-end was designed to submit cancellation requests and monitor order status, while

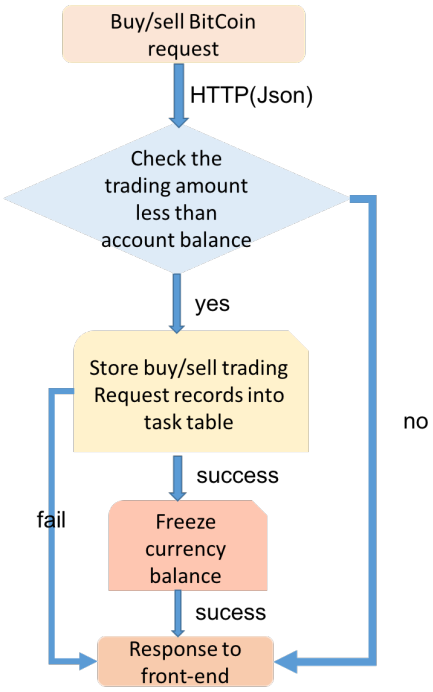

Figure 5.5: The phases of buy/sell trading processing.

the actual cancellation jobs were processed in the auto-trading engine. Finally, the feedback from auto-trading engine will update the order status in the front end. The different phases of order cancellation implementation can be illustrated as Figure 5.6.

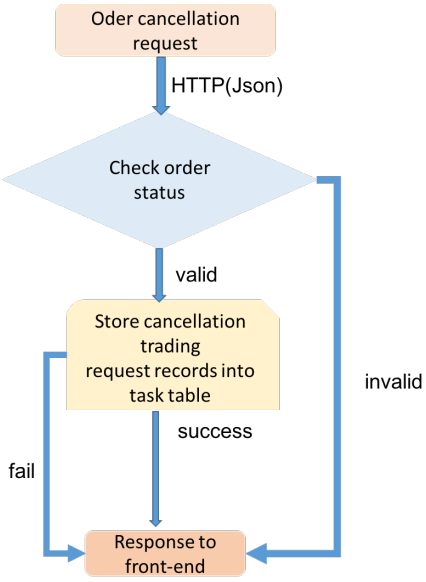

Figure 5.6: The phases of order cancellation processing.

From the figure, it shows that when an order cancellation request was submitted to the

back-end, the Order Cancellation Controller will do following tasks:

1. Query the selected order from **TradeInfo** table by using the ”id” field.
2. Check the status of the order to determine whether this order can be canceled. If the status was ”000” or ”001”, go to step2. Otherwise, return **This order cannot be canceled** to front-end to notify users the failed reason.
3. Insert an new task with type = ”cancel”, status = ”000” into **TradeTask** table. If inserted successfully, return **Cancellation request submit successfully**, otherwise return error message.

5.2.3.3 Order Searching Service

For order searching service, a Trading Order Searching Controller was designed in back-end deal with trading order searching request from the front-end. The processing logic can be described as following steps:

1. Query trading order records from **tradeInfo** table by using time period and the requested user’s account.
2. Return the results list to front-end to display.

5.3 Auto-trading Engine

In this section, business requirements and detail implementation are introduced subsequently to explain the design process of Auto-trading Engine.

5.3.1 Business Requirements

The basic trading rules for Bitcoin transactions are the same as other online trading platforms. The Buy request will try to find the lowest asking price from the satisfied Sell requests and select the earliest Sell request if multiple sellers available with the same lowest asking price. Similarly, the Sell requests will find the highest biding price first from satisfied Buy requests. If multiple buyers with the same highest biding price are available, chose the earliest one.

5.3.2 Implementation of Auto-trading

To meet these business requirements, instead of doing trading instantly when received trading requests from Front-end, the Auto-trading engine was designed to be a timing task in the back-end by using the akka.actor.Scheduler in Scala with a constant time interval. By this design, the submitted tasks will be processed sequentially to guarantee consistency. As a result, one trading request will not be dealt multiple times, which means the concurrent issues of trading requests can be avoided. The time interval was set to be 30s at this stage, and will be further discussed in evaluation section. The implementation of this timing task can be specified as below:

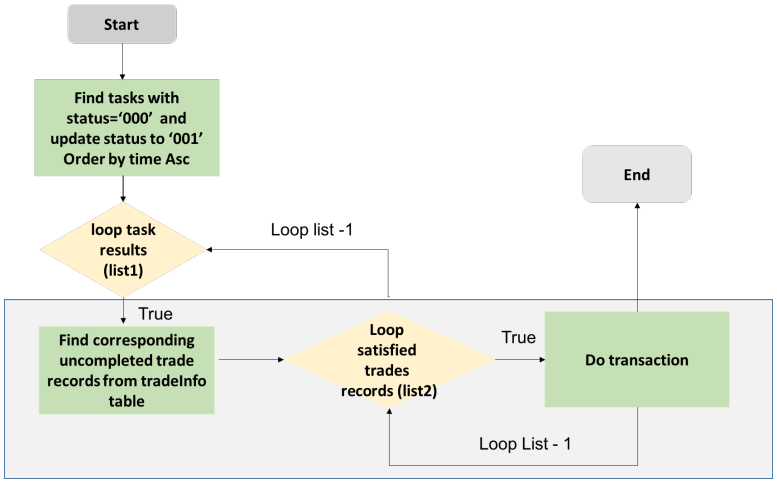

Figure 5.7: The phases of Auto-Trading processing.

As the above flow diagram showed, the Auto-trading engine was implemented by two embedded loops and task status updating process. In order to do tasks matching, all submitted tasks are marked with three different types (Buy, Sell and Cancel). Before introducing the detail process, the tasks saved in **tradeTask** Table and **tradeInfo** table are clarified clearly as below.

- **TradeTask Table:**

  1. Tasks with Pending State: new submitted buy/sell/cancel tasks (**status value: 000**);
  2. Tasks with Being-Processed State: tasks that are being processed by the Auto-trading system (**status value: 001**);
  3. Tasks with Cancel Failed State: tasks that are failed canceled (**status value: 004**);

- **TradeInfo Table:**

    1. Tasks with unsolved State: tasks that fail to complete in the auto-trading system (**status value: 001**);
    2. Tasks with solved State: tasks that are already completed in the Auto-trading system (**status value: 002**);
    3. Tasks with canceled State: tasks that are successfully canceled in the Auto-trading system (**status value: 003**).

The detail processing steps in the auto-trading module are introduced as below:

1. The submitted new buy/sell/cancel tasks will be inserted into the **tradeTask** Table with pending state;
2. Every 30 seconds, all tasks with pending state will be updated to Being-processed state and return a sorted task list (List 1) by submitting time with ascending order;
3. Loop List 1 acquired from step 2. For each task, a corresponding searching job will be processed to find matched unsolved tasks(with status="001") in **tradeInfo** Table. For each searching job, only matched task type will be selected. This means that the buy task will only find sell task records, the sell task will only find buy task records, and the cancellation task will find the relevant trading task which the current user want to cancel.

    - For example, when a buy task submitted with biding price 200GBP to buy 1 BitCoin, then all unsolved sell tasks in **tradeInfo** Table with asking price less or equal than 200GBP and with status="001" will be selected.

    The searching results will return a new task list (List 2) sorted by price and placing time with priority in price by ascending order.
4. Loop List 2 acquired from step3 until current task in List 1 completed or loop finished. Different operations will be activated for different task types.

    - For buy/sell tasks, the processing operations includes inserting records of transaction into **tradeOrders** table, updating account currency balance to **userAccounts** table, inserting currency balance updating flow into **accountUpdateFlow** table, updating task status and adding the unfinished trading task into **tradeInfo** table for future transactions. After this step, this trading

task and the matched trading tasks in List 2 will be updated to **solved** status. If the task was unfinished or partially-finished, the trading task with the unfinished amount will be update into the **tradeInfo** Table with unsolved status.

- For cancellation tasks, the first operation is to check the canceled order status, if the order was ”finished” or ”canceled”, this cancellation tasks will be updated to ”failed” with status = ”004”. Otherwise, if the order status was valid, the next operation is updating its status to be **canceled** and the cancellation task status to be **solved**, and then updating account currency balance by moving the trading amount from frozen amount to the account balance.

5. The last step is finishing this timing task and repeat next iteration. In order to prevent concurrent errors in the Auto-trading engine, the tasks are processed sequentially instead of parallelization. ”Await” function has been used to achieve this goal.

   It is important to mention that, for each timer task iteration, the tasks with being processed state will still be processed if it takes longer than specified interval (e.g. 30s), because the next timer task execution thread will be put into a queue and start after the previous one completed.

5.4 Summary

Through above implementation process, the back-end of this online trading platform is successfully developed. All functions required in front-end interfaces have been realized effectively. In addition, the auto-trading system is optimized after numerous design iterations to maintain efficiency and accuracy simultaneously for large amount of trading orders, which is the core function for an trading platform. Although prototype tests were conducted through the whole front-end and back-end design process, a scientific and rigorous evaluation is still needed for web application to reflect its usability and performance. The next chapter will present the full evaluation process and analysis results.

Chapter 6

Evaluation and Results

For web application development, the evaluation is a critical step to guarantee the quality of the application. In this project, a set of test cases have been designed and carried out. Different levels of testing strategies were applied to make sure that this system was well-designed and accurately implemented.

This chapter will focus on two main evaluation methodology: Usability Evaluation, Performance Evaluation. Each section will contain testing methods introduction and results discussion.

6.1 Usability Evaluation and Results

Usability is the degree to which a web application can be used by users to achieve quantified objectives with effectiveness, efficiency and satisfaction in a quantified context of use[14]. In this project, the evaluation factors of usability includes User Interface (UI) usability testing and Basic functionalities testing .

6.1.1 UI Usability Testing

Nielsens usability heuristics was selected to be the evaluation method for the User Interface (UI) usability. The results are presented in the table below.

According to the results of Nielsens usability heuristics evaluation displayed in the Figure 6.1, in spite of some cosmetic problems in terms of flexibility and help docu-

Heuristic	Scale
Visibility of the system's status: Is the interfaces of the platform always informed of what is going on?	**0**
Match between system and the real world: Is the information presented in logical order? Is the operation in the system meets user's expectations derived from their real-world experiences?	**0**
User control and freedom: Are the backward steps for each functionality interface are possible ? How easily is the user able to recover from errors ?	**0**
Consistency and standards: Are the graphic elements and terminology maintained across similar platform ?	**0**
Error prevention: How helpful is the system in preventing the users from making errors ?	**0**
Recognition rather than recall: Does the user need to remember the information from another part of the dialog when he proceeds to another stage of the system?	**0**
Flexibility and efficiency of use: Does the system support users to customize or tailor the interface to suit their needs ?	**1**
Aesthetic and minimalist design: Does the system include the necessary elements that are the simplest components to the current tasks ?	**0**
Help users recognize, diagnose and recover from errors: Does the system error messages display in plain language ?	**0**
Help and documentation: Is there a need to introduce additional documentation into the system?	**1**

Notes:
0 = I don't agree that this is a usability problem at all;
1 = Cosmetic problem only: need not be fixed unless extra time is available on project;
2 = Minor usability problem: fixing this should be given low priority;
3 = Major usability problem: important to fix, so should be given high priority;
4 = Usability catastrophe: imperative to fix this before product can be released.

Figure 6.1: The results of heuristics testing

mentation, the usability of the UI of this platform has achieved the requirements of 10 Nielsens heuristics for web application. Due to the limitation of time, these cosmetic problems are planned to be fixed in the future work.

6.1.2 Basic functionalities Testing

The basic functionalities tests were applied to evaluate whether the basic functionalities such as account management services, finance management services and trading services perform in successful manner or not. A set of testing cases were designed to check if the system works as it is expected to. The test results were summarized in the following evaluation table. It can be seen from the Figure 6.2 that basic func-

Module	Functionalities	Requirements	Mostly Satisfied	Satisfied	Mostly Unsatisfied	Unsatisfied
Account Management	Register/Login Service	Check new user can register		✅		
		Check registered user can login		✅		
		Check stored password is encrypted		✅		
	Password Resetting Service	Check password can be updated		✅		
Finance Management	Account Balance Display Service	Check account balance can be display correctly		✅		
	RMB Deposit/Withdraw Service	Check RMB deposit can deposit money correctly		✅		
		Check account RMB balance is updated correctly		✅		
	GBP Deposit/Withdraw Service	Check GBP deposit can deposit money correctly		✅		
		Check account GBP balance is updated correctly		✅		
	BTC Transfer Service	Check BTC deposit can deposit money correctly		✅		
		Check account BTC store is updated correctly		✅		
	Account Transaction Records Searching Service	Check all satisfied transaction records can be display		✅		
		Check searching conditions work		✅		
Trading Management	Buy/Sell Trading Service	Check Buy Bitcoin operation can submit		✅		
		Check freeze account currency balance correctly		✅		
		Check Sell BTC operation can submit		✅		
		Check freeze BTC store correctly		✅		
	Order Cancellation Service	Check customer order records can be searched		✅		
		Check order cancellation can work		✅		
		Check cancellation history can be searched		✅		
	Order History Searching Service	Check all satisfied transaction records can be display		✅		
		Check searching conditions work		✅		

Figure 6.2: The report of functionalities testing

tionalities of this system have been perfectly implemented and successfully meet all functionalities testing requirements.

6.2 Performance Evaluation and Results

This section will first investigate the performance of this trading platform for large amount of concurrent HTTP requests. Then the processing ability of Auto-trading engine will be evaluated. A professional testing tool called Apache JMeter which has been introduced in Chapter 2 was used to complete these testing tasks.

6.2.1 Response time for HTTP Requests

Considering that trading requests are the most frequent and concurrent operations, the evaluation of large amount of trading requests in a short time can represent the performance of this trading platform for handling HTTP requests. The load tests are introduced by following three parts:

- Testing cases
- Testing scripts
- Testing results

Test cases: Multi-thread requests were applied through JMeter to simulate concurrent and high load trading requests. The load testing of trading service was separated into two parts: one is to test the maximum number of concurrent requests that this platform can support, the other one is to investigate the trend of response time for trading requests in different load tests.

	Threads(Num)	Ramp-Up(Period)	Loop(Count)	User accounts(Count)
1	50	0	10	100
2	100	0	10	100
3	200	0	10	100
4	300	0	10	100
5	400	0	10	100

Table 6.1: The test cases for trading load test.

Threads(Num): defines the number of concurrent threads that used to simulate for the execution.
Ramp-Up(Period): defines the amount of time that Jmeter should take to get all the threads sent for the execution.
Loop Count: defines the number of times to execute the Performance Test.
Users(Count): defines the number of user's accounts that used to do the performance test.
In order to obtain the maximum number of concurrent requests the system can support, five test cases were build in JMeter as the above table 6.1 showed. The value of Ramp-Up was set to 0, it means that testing script will initiate all threads immediately at the beginning of the test execution, and therefore it will put very severe load on the application at once.

For the investigation of response time in different load tests, 78 JMeter test cases (shows in table 6.2) with different parameters setting were executed. Two parameters were variables in this testing: one is the number of threads, the other one is the period of Ramp-up. Two parameters were constant, including Loop count and the number of user account. 100 different user accounts were applied in this testing to sent Bitcoin trading requests. In addition, each testing case was execute 10 times to make sure the quality of performance testing.

Parameter	Test case values
Threads(number)	10,50,100,150,200,250,300,350,400,450,500,550,600
Ramp-up period(s)	1,2,4,6,8,10
Loop Count	10
User Accounts	100

Table 6.2: The test cases for trading load test.

Test scripts: Two test scripts were created to complete this testing: the first script is to get trading information automatically from a TXT file, the other one is to check whether the trading is success. The following Figure 6.3 and 6.4 show test scripts setting in JMeter.

Test Results: From the first testing results (Figure 6.5) , it can be seen that the maximum number of concurrent requests this online trading platform can support is around 400. When 400 trading requests submitted to the trading service at the same time, the

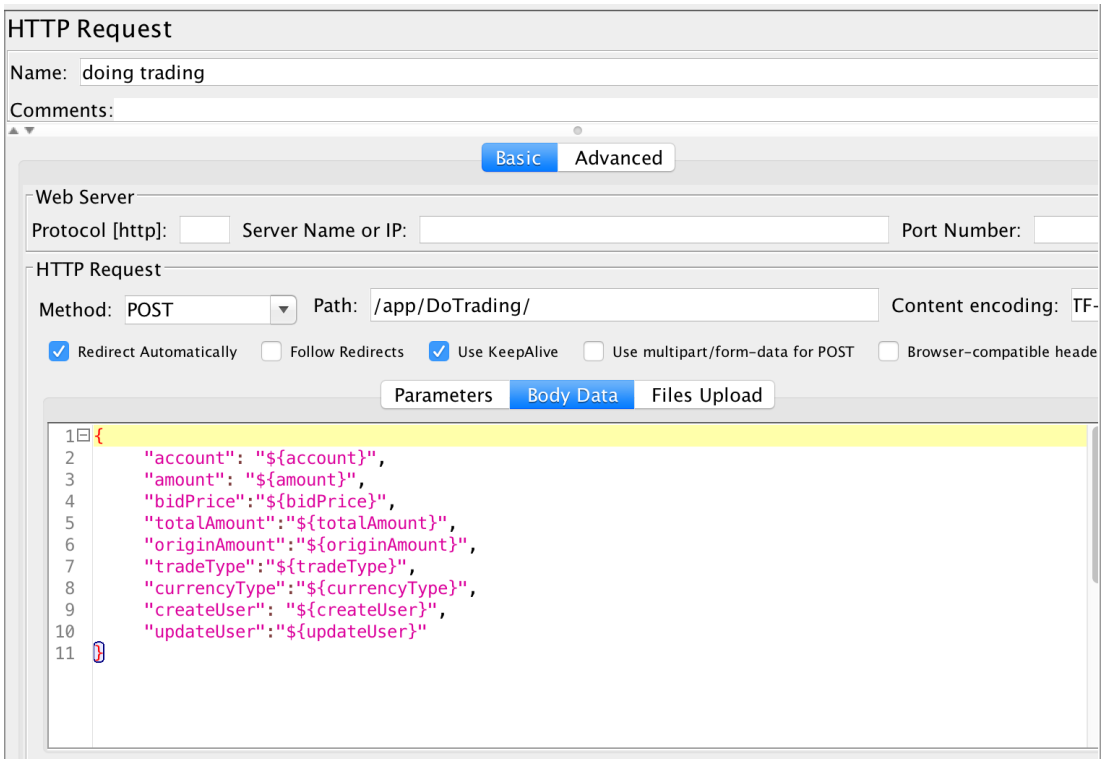

Figure 6.3: The test script for getting transaction information.

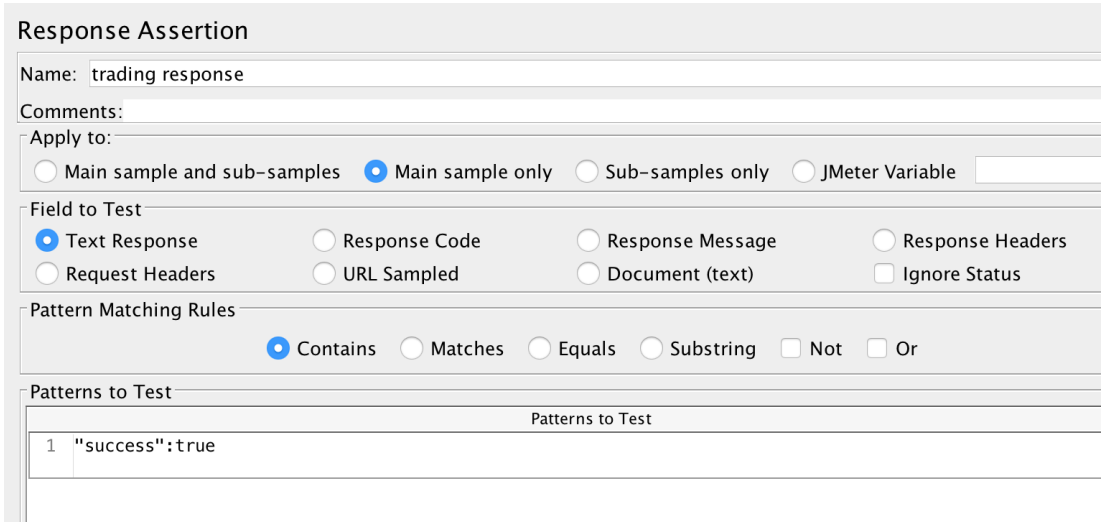

Figure 6.4: The test script for checking the result for trading test.

error rate reached 0.03%. Meanwhile, the CPU usage was over 90%, it means that the load of server reached the peak state. The average throughput of system was around 380 request per second, which is fast enough for this platform.

The following heatMap Figure 6.6 can reflect the trend of average response time of each trading request in different load tests. From the heatMap result showed above, it

Label	# Samples	Average (ms)	Median (ms)	90% Line (ms)	95% Line (ms)	99% Line (ms)	Min (ms)	Max (ms)	Error %	Throughput	Received KB/sec	Sent KB/sec
1	500	174	129	420	478	572	7	679	0.00%	266.80896	51.07	123.57
2	1000	210	214	302	325	366	21	449	0.00%	406.00893	77.71	188.04
3	2000	470	441	762	856	1391	6	1536	0.00%	347.64471	66.54	161.01
4	3000	641	624	1050	1138	1723	25	2840	0.00%	385.60411	73.81	178.59
5	3500	689	668	1118	1234	1415	8	1887	0.00%	404.76466	77.47	187.46
6	4000	875	906	1236	1360	1570	3	2137	0.03%	383.61945	73.65	177.63

Samples - The number of completed transactions in this scenario
Average - The average time of response
Median - The median is the time in the middle of response.
90% Line - 90% of the samples took less than this time.
95% Line - 95% of the samples took less than this time.
99% Line - 99% of the samples took less than this time.
Min - The shortest response time for the samples.
Max - The longest response time for the samples.
Error% - Percent of requests with errors
Throughput - the Throughput is measured in requests per second/minute/hour.
Received KB/sec - The throughput measured in received Kilobytes per second
Sent KB/sec - The throughput measured in sent Kilobytes per second

Figure 6.5: The results of trading load testing.

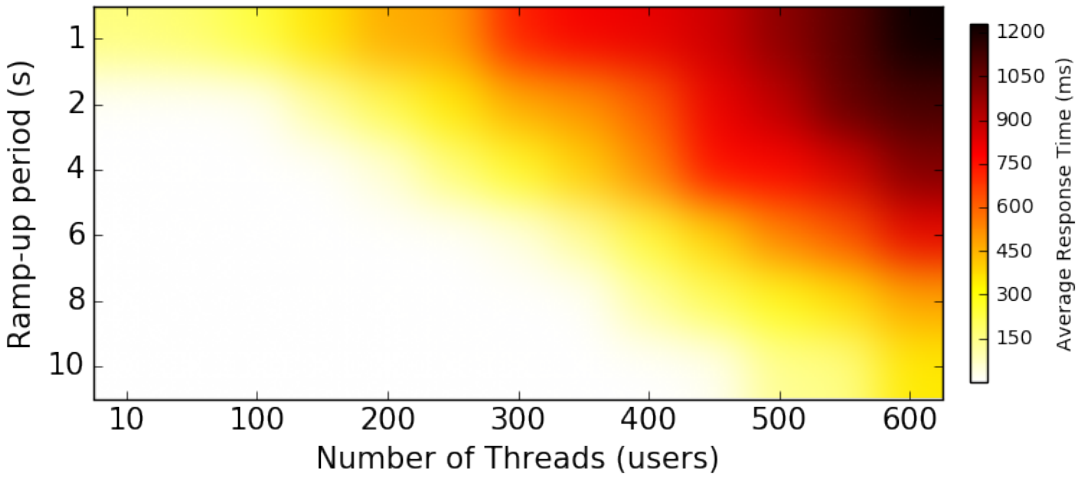

Figure 6.6: The heatMap of trading load testing.

can be summarized that, for same Ramp-up period, the average response time increased with the increase of concurrent threads. Similarly, with the same concurrent threads number, the response time increased for lower Ramp-up period. It is clearly that the processing time of trading requests will rise for higher concurrent load.

The average response time was fast (under 0.1 s) for white area in the HeatMap. The performance in yellow and red area are also acceptable, with average response time still below 1 s. However, When the number of concurrent requests increased to 600 and the Ramp-up period was set between 1 to 3, the response time exceeds 1 s per request. Users may feel a bit slow about this system for this response time.

6.2.2 Processing Ability of Auto-trading Engine

In order to evaluate the processing ability of Auto-trading engine, both accuracy and processing speed of this trading system need to be reflected. This evaluation was achieved through JMeter to submit a certain number of trading requests to Front-end interfaces, and then these requests will be processed in the Auto-trading engine. The processing time for all trading orders in the Auto-trading engine have been recorded to reflect to process speed. The trading records can be used to assess the accuracy. Sufficient balance for GBP currency and Bitcoin have been put in the 100 testing accounts to avoid negative balance error.

Detail test cases are showed in following Table 6.3. For each certain amount of trading requests, half of them are set to be Buy requests and other half set to be Sell requests. Ideally, these requests can be all completed within one iteration.

Tasks(Num)	Period(s)	Tasks(Num)	Period(s)
1000	5	100	30
1000	30	1000	30
1000	60	10000	30
1000	90	20000	30
		30000	30
		40000	30
		50000	30

Table 6.3: The test cases for auto-trading engine evaluation.

Test script: The script shows below was used to record the processing time of auto-trading.

```
def processTime[R](function: => R): R = {
  val t0 = System.nanoTime()
  val result = function //function is used to do auto-trading
  val t1 = System.nanoTime()
  println("Cost time: " + (t1 - t0) + "ns")
  result
}
```

Test Results: The total processing time and average processing time of trading orders from above test cases are summarized in Figure 6.8.

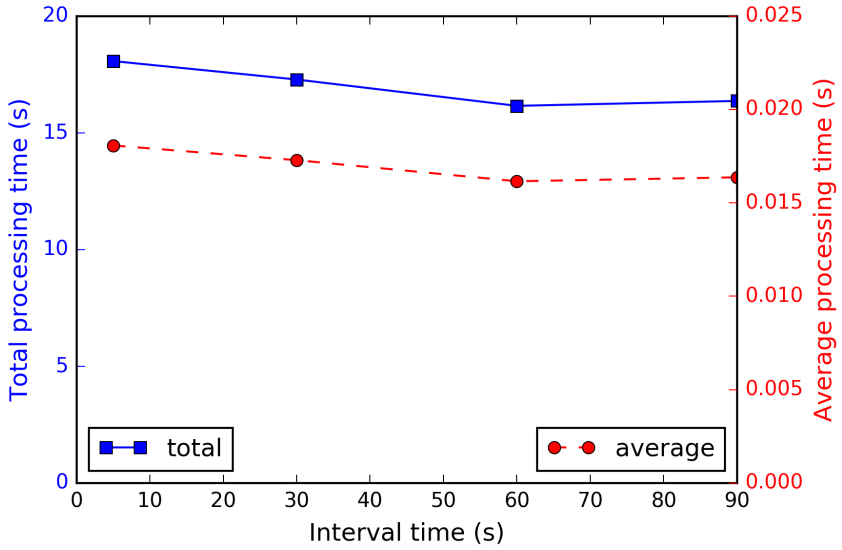

Figure 6.7: The test results of Auto-trading engine with different interval time.

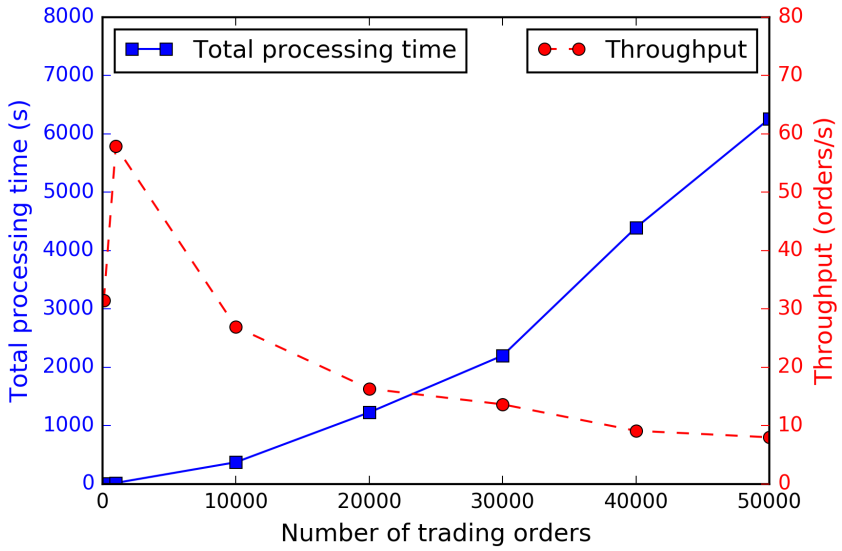

Figure 6.8: The test results of Auto-trading engine with different trading orders.

- **Accuracy Evaluation:** According to above test results, all trading orders have been completed successfully. The trading records, trading task status and account balance for all test cases have been checked and shows correct outcome. For high amount of trading orders, the pre-defined iteration period (30 s) is not

sufficient to process all orders, but as expected in the implementation, the rest orders have been continually processed and postponed current iteration time. Even for high load orders (50,000), the system still be able to deal with all orders correctly without any error occurred, and CPU usage kept below 40% for all test cases. It is supposed that this sequential trading processing design guaranteed the stability and high accuracy of this system.

- **Processing Speed Evaluation:** From Figure 6.7, it was found that the interval time has litter affect on the processing time. In Figure 6.8, the total processing time increased significantly for high amount of trading orders, which is easy to explain. As more trading orders are inserted into the **tradeTask** and **tradeInfo** Table, more tasks will be processed one by one. As the curve showed, the throughput of this trading engine increased first, and then decreased when the number of orders reached 1,000 , then kept relative stable when the number of orders went above 30,000. The average throughput of trading engine was highly dependent on the number of processing trading orders and could decreased to 10 (orders/s) for over 30,000 orders. Since these processes are conducted in back-end, it will not affect the overall system operations.

6.3 Summary

In summary, through above evaluation and analysis, it can be concluded this online trading platform is correctly implemented with all designed functionalities. In addition, the system can accurately and efficiently deal with large amount of HTTP requests and trading orders. For HTTP requests, the response time is considerable fast owing to the parallel implementation with average throughput around 380 (request/s). The maximum capacity for concurrent requests is around 400. For processing trading orders, high stability and accuracy have been confirmed through a series of evaluation tests. Although the order processing capability of auto-trading engine is slow (about 10 to 50 orders/s), the total and average processing time are acceptable. Possible methods to improve the performance of auto-trading can be explored in the future work.

Chapter 7

Conclusions

7.1 Summary

In summary, a fully self-developed open source online trading platform for Bitcoin was successfully built through this project. This trading system explored a new technology architecture consisted of HTML, AngularJS, Scala, MongoDB and Play Framework. The initial pre-design stage extracted the required system goals for this project, and was modified based on the analysis of constraints. All core functionalities for this trading platform have been accurately identified and successfully realized in front end and back end.

In front-end, multiple interfaces with optimized user experience have been successfully developed, including account management, finance management, trading management and searching functionality etc. A series of front end validations were rigorously implemented. These interface validations can significantly improve the usability and performance of the platform, because it can prevent users from making system and trading errors. Moreover, a dashboard was implemented to display the real-time updated Bitcoin trading records.

In the back-end, several Data schema in mongoDB have been created and structured to support the later implementation of core functionalities. A Controller module was developed for each required function to process the business logic and incoming requests, manipulate data in mongoDB. All business requests from front-end can be processed correctly after implementing the back-end design. In addition, an advanced auto-trading engine that designed to process trading orders as a time schedule jobs has

been elaborately designed and optimized to solve concurrent issues and maintain high efficiency.

A series of evaluation tests have been designed to test the feasibility and efficiency of this online trading platform. According to the testing results, all interfaces on the platform are correctly displayed and all basic functionalities perfectly reached the system requirements. Furthermore, the system performed well in terms of concurrency and load testing. The auto trading engine can handle transactions correctly without concurrent errors, and the speed of processing is acceptable. It can be concluded that this online trading platform for Bitcoin can run correctly and smoothly for a large number of users.

7.2 Critical Analysis

Through the whole developing process, the performance and possible weakness of this trading platform are critically discussed here.

For Front-end implementation, AngularJS is proved to interact with back-end high efficiently. Play framework is found to efficiently process asynchronous HTTP requests. The HTTP requests can be processed in parallel without concurrent issues. With a laptop server, the system is already capable of dealing large amount of concurrent requests (400) in a short time.

In the evaluation part, due to limitation of time, the interface functionalities testing is conducted manually, which is not rigorous enough. A better choice would be using automated UI test tools.

For auto-trading engine, the time schedule job module in Scala enables the system to process trading orders in the Back-end without affecting other system operations. This auto-trading engine is designed to process requests sequentially to avoid concurrent issues and system errors. The evaluation results confirmed the accurate performance of this trading engine.

However, the processing ability and speed is still not sufficient for real world application. More powerful server would improve its performance. Other optimization methods can be applied in the future.

The data model based on MongoDB can manipulate data efficiently and precisely.

However, the main advantages of MongoDB on dealing with no-relational data have not been reflected in this project at this stage. In the future, the system logs and other no-relational data files can be stored into MongoDB efficiently.

In this project, all transfer and trading requests are virtually simulated without interacting with real third-parties. In the trading part, only one kind of trading requests with constant bid/ask price and amount of Bitcoin is available, which is not sufficient for real world application. In the future, the connection between Bitcoin network and finance institutions should be build to support real transactions.

7.3 Future Work

Based on above critical analysis, several directions can be taken to expand this project for better overall performance and real world application.

- For real world application, this trading platform should be connected to third party finance institutions and linked to Bitcoin network for real transactions. Furthermore, the API which used to connect with Bitcoin network and finance institutions need to be carefully designed in order to get high performance and guarantee security.
- To further improve the processing ability of this trading platform, the auto-trading system is possible to be re-designed for getting order information from cache instead of visiting database repeatedly. Possible approach is searching the earliest trading orders (e.g. the first 100 orders) from database each time, and storing these records into a cache such as Memcached (a distributing memory caching system) to be processed. This method can save the time that cost on database searching to improve the processing performance.
- The trading services can be expanded to support various types of Bitcoin trading, for example, buying a fixed number of Bitcoins without a constant biding price.
- The UI can be further improved for better user experience and integrated with more practical functions.
- Implementing the system on multiple servers to enhance its processing ability, use MongoDB to do project partition.

Bibliography

[1] Scala Description. http://www.scala-lang.org, 2001.

[2] Security at mtgox much worse than originally imagined. https://falkvinge.net/2014/03/11/just-how-abysmal-was-gox-security-anyway, 2014. Rick Falkvinge.

[3] The Wall Street Journal. https://blogs.wsj.com/briefly/2014/02/25/5-things-about-mt-goxs-crisis/, 2014. Vigna, Paul (2014-02-25).

[4] Apache JMeter. http://jmeter.apache.org/index.html, 2015.

[5] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[6] Simon Barber, Xavier Boyen, Elaine Shi, and Ersin Uzun. Bitter to betterhow to make bitcoin a better currency. In *International Conference on Financial Cryptography and Data Security*, pages 399–414. Springer, 2012.

[7] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[8] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[9] Brad Dayley. *Node. js, MongoDB, and AngularJS Web Development*. Addison-Wesley Professional, 2014.

[10] Alan Dix. Human-computer interaction. In *Encyclopedia of database systems*, pages 1327–1331. Springer, 2009.

[11] Brad Green and Shyam Seshadri. *AngularJS*. " O'Reilly Media, Inc.", 2013.

[12] Philipp Haller and Frank Sommers. *Actors in Scala*. Artima Incorporation, 2012.

[13] John Hunt. Play framework. In *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, pages 413–428. Springer, 2014.

[14] W Iso. 9241-11. ergonomic requirements for office work with visual display terminals (vdts). *The international organization for standardization*, 45, 1998.

[15] Nilesh Jain, Priyanka Mangal, and Deepak Mehta. Angularjs: A modern mvc framework in javascript. *Journal of Global Research in Computer Science*, 5(12):17–23, 2015.

[16] T.P. Lemke and G.T. Lins. *Soft Dollars and Other Trading Activities*. Securities law handbook series. West, 2011.

[17] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[18] Shamkant B Navathe and Mario Schkolnick. View representation in logical database design. In *Proceedings of the 1978 ACM SIGMOD international conference on management of data*, pages 144–156. ACM, 1978.

[19] Klaus Purer. Php vs. python vs. ruby–the web scripting language shootout. *Vienna University of Technology*, 2009.

[20] Stephen Turner. Security vulnerabilities of the top ten programming languages: C, java, c++, objective-c, c#, php, visual basic, python, perl, and ruby. *Journal of Technology Research*, 5:1, 2014.