Ministry of Education of the Republic of Belarus

Institution of Education
BELARUSIAN STATE UNIVERSITY
OF INFORMATICS AND RADIOELECTRONICS

Faculty of Information Technologies and Control

Information Technologies in Automated Systems Department

Diploma project submission permitted
The Head of the Information Technologies in Automated Systems Department

_____A.A. Naurotsky

EXPLANATORY NOTE
Diploma Project

Automated Mobile Payment System

BSUIR 1−53 01 02 01 008 DP

Student	Tu Xinyuan
Supervisor	A.F.Trofimovich
Advisors:	
- *from the Information Technologies in Automated Systems Department*	A.F.Trofimovich
- *for the Economic feasibility study*	I. V. Smirnov
Standards Compliance Inspector	N. V. Batin
Reviewer	

Minsk 2022

ABSTRACT

AUTOMATED MOBILE PAYMENT SYSTEM : Diploma Project / TU Xinyuan – Minsk : BSUIR, 2022, – Explanatory Note – 77 p., drawings – 6 A1 sheets.

The diploma project deals with the design of the computerized system for the automated mobile payment system. The essay includes the structure analysis of mobile payment system, related data flows and existing comparisons of analogous computerization systems. Solutions has been raised for computerization of several tasks in the system, such as product demand and description management, user's information management, payment and transfer processing and bill checking. The solutions cover proper algorithms, illustration of data flows, database design, system implementation and deployment.

The software has been designed for implementation of these tasks. Programming and data management tools used for software implementation include IntelliJ IDEA, Oracle Java, Android Studio, Kotlin and MySQL. Software operation modes for these categories of users as well as the administrator's mode for software installation and setup have been provided. User's and administrator's manuals have been prepared.

The economic feasibility study has been carried out, confirming the project's cost – effectiveness. The expected economic effect resulting from the designed software application has been calculated.

CONTENTS

INTRODUCTION

Mobile payment, also named mobile money, mobile money transfer and mobile wallet, generally refers to payment services executed on the mobile terminals. Compared to the traditional payment by cash, cheque and credit cards, mobile payment shows advantages in a wide range of applicability, security and convenience. Furthermore, a better interactive experience is provided in mobile payment. In terms of anti – money laundering and fund trackability, the mobile system is more effective.

In general, there are four main models for mobile payments[1] :

- Bank–centric model.
- Operator–centric model.
- Collaborative model.
- Independent service provider(ISP) model.

In the first and second models, a bank or the operator is the central node in the mobile payment system, managing the transactions and distributing the property rights. In the collaborative model, the financial intermediaries and telephonic operators collaborate in managing tasks and cooperatively share the property rights. In the ISP model, a third–party agency of high confidence operates as an independent and neutral intermediary between financial agents and operators. Apple Pay or PayPal are the typical ISP mobile payment provider.

A mobile wallet is an app that contains the user's debt and credit card information, letting them pay for goods and services digitally via their credit cards directly or indirectly. Notable mobile wallets include:

- Alipay.
- Apple Pay.
- Google Pay.
- WeChat Pay.
- Samsung Pay.

For instance, Alipay, a third – party mobile payment platform established in China, is a combination of the Operator – centric model and ISP model, which provide an internal financial system and external bank communication services. Google Pay and Apple Pay are purely ISP providers and only act as invoking services provided by bank systems to make transactions.

In Belarus, the mobile payment development is at the initial stage, with few proportions of usage. Still lot of people prefer to use traditional payment. With the digitization tendency, mobile payment will replace traditional payment gradually. Small – scale business runners can enjoy the benefits of mobile payment since

mobile payment does not rely on the POS−alike machine, which is the necessary transaction component.

Considering factors above, now it’s the perfect time to develop a mobile payment application in Belarus. This project aims to create a new mobile android app and its corresponding backstage management system on web page. The process of the app is described roughly as below.

A user can log in to the system by their account. Inside the system, services including exporting money, importing money, transferring, receiving and paying are provided. By providing valid merchant information, a user can register to be a merchant, unlocking relevant business functions, including unlimited payment receiving and invoking the system's API.

This essay illustrates the detailed design of the Automated Mobile Payment System. All the necessities are included.

1 ANALYSIS OF SUBJECT AREA

1.1 Development of Payment

1.1.1 Bartering and Livestock

Ages ago, there was a time when standard money did not exist, and transactions were made in other forms. The earliest form is bartering and livestock. Barter is one of the types where goods or services are exchanged for a certain amount of other goods or services (see figure 1.1); no standard currency is involved in the transaction. It can be bilateral or multilateral trade. One common form of barter during colonial times was tobacco. Also, bushels of grain and wampum were popular forms. Barter trade is common among people with no access to a cash economy, in societies where no monetary system exists, or in economies suffering from a volatile currency (as when hyperinflation hits) or a lack of currency [3].

One prime disadvantage of using bilateral is that it heavily depends on the mutual traders' level of wants. In detail, if either of the traders is not interested in the items to be exchanged, the transaction may end up failing.

Figure 1.1 - Bartering and Livestock Example

1.1.2 Precious Metal Coins

Ancient civilizations used to use beads and shells as coins, and eventually, they began using precious metals to make coins. People in the ancient civilization of Lydia were among the first to use coins made of gold and silver, and this currency was both valuable and easily portable[4].

However, the shortcoming of the coin type is obvious – it weighs when carrying a large amount of money. Also, due to the rarity of raw materials in making coins, the coin cannot be widely distributed.

1.1.3 Leather Money

Leather used to be the material for currency. It can date back to ancient China where white leather made of deerskin was utilized for banknotes. These notes were large compared to the bills used in today's society. Leather money could have been as significant as one–foot squares of deerskin

1.1.4 Paper Money

After a certain period of years, when Chinese developed mature paper–making techniques, paper money started to replace leather money (see figure 1.2). In addition, challenges came in the forms of both inflation and the production of the currency.

One of the inconveniences of paper money in daily lives is the complexity of giving change if not with just an amount. A large amount will be divided into smaller but hard–to–collect banknotes, which can be annoying for customers.

Figure 1.2 - Chinese Paper Money

1.1.5 Credit Cards

Credit cards started to be used in the 20th century. In its non–physical form, a credit card represents a payment mechanism which facilitates both consumer and commercial business transactions, including purchases and cash advances. A credit card generally operates as a substitute for cash or a check and most often provides an unsecured revolving line of credit. The borrower is required to pay at least part of the card's outstanding balance each billing cycle, depending on the terms as set forth in the cardholder agreement. As the debt reduces, the available credit increases for accounts in good standing. These complex financial arrangements have ever–shifting terms and prices. A charge card differs from a credit card in that the charge card must be paid in full each month.

In physical form, a credit card traditionally is a thin, rectangular plastic card. The front of the card contains a series of numbers that are representative of various items such as the applicable network, bank, and account. These numbers are generally referred to in aggregate as the account number or card number. A

magnetic stripe, often called a magstripe, runs across the back of the card and contains some of the account's information electronically. The back of the card also contains a cardholder signature box [5].

People can access their funds by tapping or inserting their credit cards at merchant terminals and service providers (figure 1.3).

Figure 1.3 - Credit Card Payment

1.1.6 Mobile Payments

Mobile payment (also referred to as mobile money, mobile money transfer, and mobile wallet) generally refer to payment services operated under financial regulation and performed from or via a mobile device. Instead of paying with cash, cheque, or credit cards, a consumer can use a mobile to pay for a wide range of services and digital or hard goods. Although the concept of using non–coin–based currency systems has a long history ,it is only in the 21st century that the technology to support such systems has become widely available.

Now mobile payment is undergoing a revolutionary in changing world transaction patterns.

1.2 Organizational Structure of Payment System

Minterzberg (1972): Organizational structure is the framework of the relations on jobs, systems, operating process, people and groups making efforts to achieve the goals. Organizational structure is a set of methods dividing the task to determined duties and coordinates them. Hold and Antony (1991): Structure is not a coordination mechanism, and it affects all organizational process. Organizational structure refers to the models of internal relations of organization, power and relations and reporting, formal communication channels, responsibility and decision making delegation is clarified. Amold and Feldman (1986): Helping the information flow is one of the facilities provided by structure for the organization (Monavarian, Asgari, & Ashena, 2007). Organizational structure should facilitate

decision making, proper reaction to environment and conflict resolution between the units. The relationship between main principles of organization and coordination between its activities and internal organizational relations in terms of reporting and getting report are duties of organization structure (Daft, Translated by Parsayian and Arabi, 1998)[2].

1.2.1 The Importance of Organizational Structure

The Importance of Organizational Structure:

- Clear definition of authority, responsibility relationship facilities better understanding of the objectives and the policies of the enterprise.
- Organizational structure lays down both channels and the patterns of communication. It facilitates proper administration.
- It helps to coordinate activities of the component parts in order to facilitate the realization of the goals of the organization.
- It helps in growth and diversification of the activities of an organization.
- Workers, participation in organization increases their cooperation and improves their will to work. It stimulates initiation and creative thinking.
- Implementation of policies and the achievement of the goals become easier.
- It prevents duplication of functions and makes it possible to achieve maximum production with minimum efforts.

1.2.2 The Organization Chart of Mobile Payment System

It is usual for the payment system structure to be depicted in the form of an organization chart. Its organization chart can be handy in providing a pictorial presentation of the structural framework of the roles and its main area of activities. It is helpful, for example, as part of a staff induction manual. The chart may also be used as a basis for structure analysis and review, training and management succession, and formulating changes.

An organization chart may show, at a given point in time, how work is divided, spans of control, the levels of authority, lines of communication, and formal relationships. Nevertheless, charts vary greatly. Some are intended to give a minimal amount of information, perhaps, for example, only an outline of the management structure of the payment system.

Every payment, needs an organizational structure to carry its operation. It is used to help divide tasks, specify the job for each role, and also define permissions. Effective job specifications will increase work productivity and efficiency. Each payment organizes workforce in different way. The following shows a the organization structure (see figure 1.3).

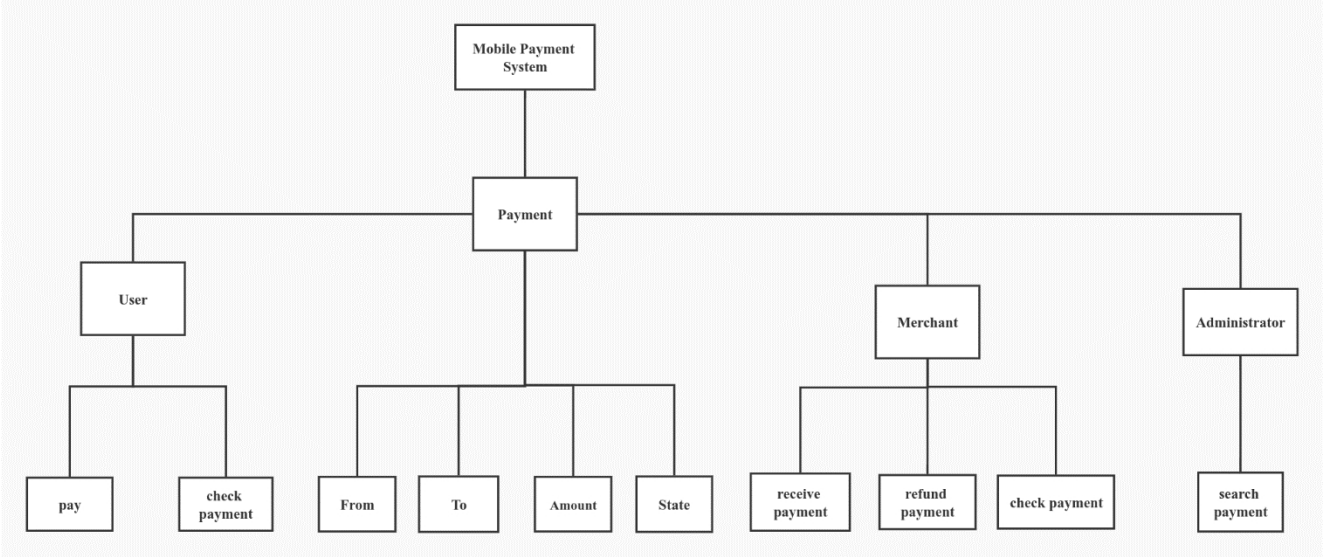


Figure 1.3 – Organizational Structure Mobile Payment System

The structure is for the system from which merchants can set payment links to receive funds, and users can pay via the payment link offered. Moreover, merchants and users can both check the payment record. One additional function, 'refund payment', is provided for merchants. For payment regulations, administrators have permission to search all the payments made.

1.3 Existing Mobile Payment System

There are varieties of mobile payment systems nowadays. Most of them are payment systems based on the ISP model, which heavily relies on credit card or debit card systems. The following section describes the existing mobile payment systems and their pros and cons.

1.3.1 PayPal

PayPal Holdings, Inc. is an American multinational financial technology company operating an online payments system in the majority of countries that supports online money transfers and serves as an electronic alternative to traditional paper methods such as checks and money orders.

The company operates as a payment processor for online vendors, auction sites and many other commercial users, for which it charges a fee. PayPal only supports password authentication, which can be leaked out accidentally.

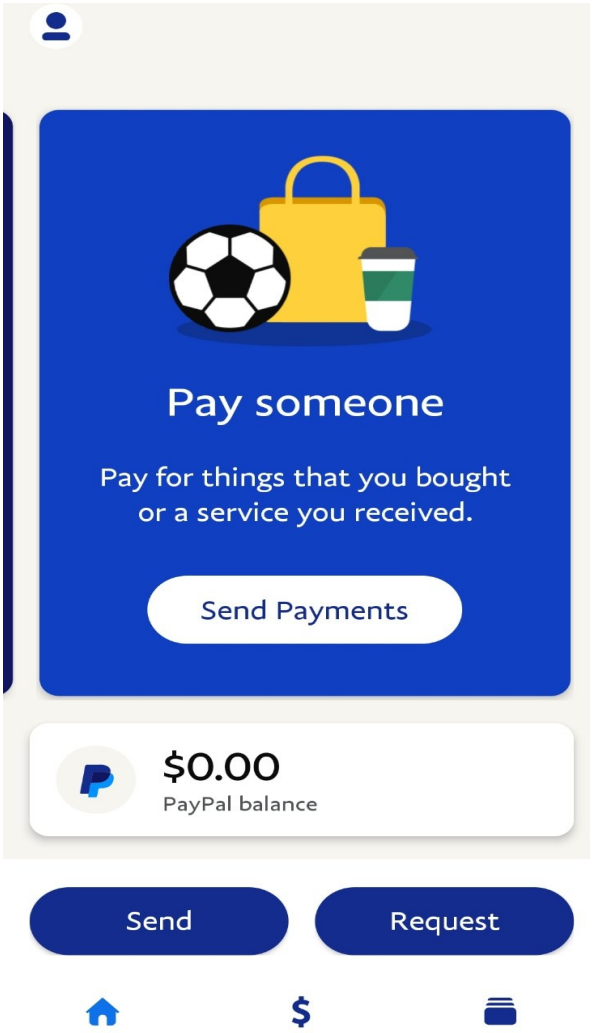


Figure 1.4 - PalPay Example

1.3.2 AliPay

Alipay is a third−party mobile and online payment platform, established in Hangzhou, China in February 2004 by Alibaba Group and its founder Jack Ma. Alipay overtook PayPal as the world's largest mobile payment platform in 2013. As of 31 March 2018, the number of Alipay users reached 870 million. It is the world's number one mobile payment service organization and the second−largest payment service organization globally. According to the statistics of the fourth quarter of 2018, Alipay has a 55.32% share of the third−party payment market in mainland China, and it continues to grow.

Alipay is conceptually similar to Apple Pay, WeChat Pay and PayPal because it overlays traditional card payment methods. The main feature of AliPay is that it supports QR Code payment. Moreover, it also supports flesh face recognition as an authentication method, which guarantees security.

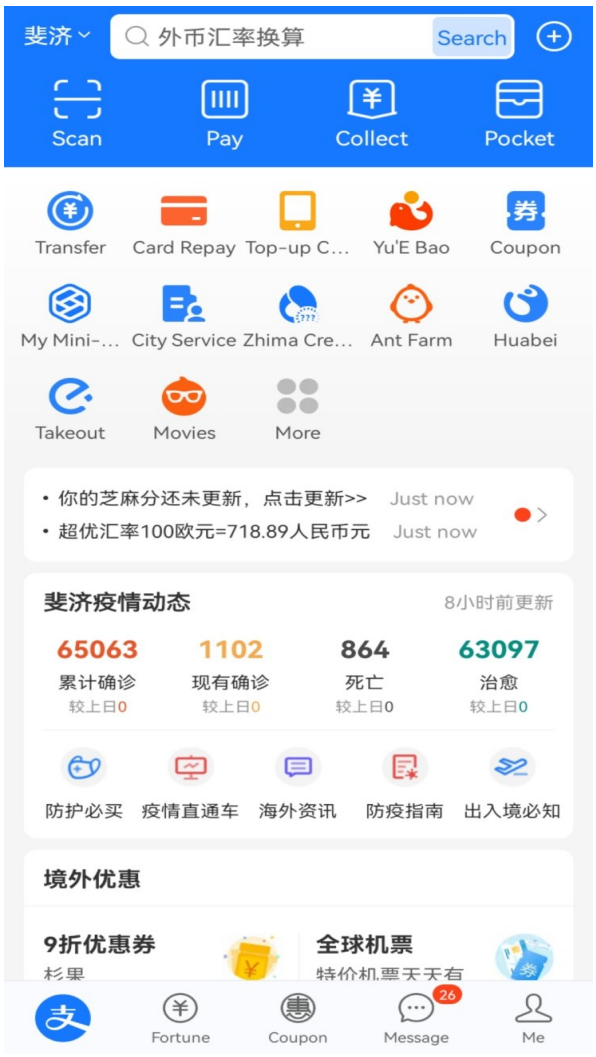


Figure 1.5 - AliPay Example

1.3.3 Drawbacks of the Existing Systems

As previously mentioned, the majority of the existing mobile payment systems are relying on credit card or debit card systems and use the protocols or interfaces provided by card organizations (Eg. Master card or Visa) or certain banks. However, when errors occur, like server crush inside the bank system or unreachable internet access, the payment fails. Obviously, the outer financial system does influence mobile payment. The mobile payment applications that may encounter such situations are Apple Pay, Google Pay ..etc.

Another mobile payment system having its own internal transaction system is Paypal. Paypal has its own balance system. It can import funds from credit cards, letting users transfer balances from other bank systems. Though it does not heavily rely on outer bank system, its ergonomics is not perfect. To make a payment, users have to manually input the Email address of the merchant, during which it is potential to input the wrong email address. Another disadvantage is that there is no effective method to determine the user's identity in the system. There may be cases where users laundry illegal funds by making transactions, which can be hard to

track. Possibly, when some business is conducted on traditional mobile payment apps, due to anonymity, taxes can be avoided by using multiple accounts for income gaining.

1.3.4 Advantages of Automated Mobile Payment System

Mobile payment consists of four types: Bank – centric, Operator – centric, Collaborative and Independent service provider(ISP). The proposed system is ISP (Independent service provider) model, which provides internal fund operation as the transaction is made. Also, the system records transactions which can be applied as a source for the audit process. Though the transaction is made online, the national tax law still works. The system stores the statistics of merchants, and if needed, it can provide tax collecting services.

As for convenience and user experience, the system provides QR Codes method for requesting and paying funds. Compared to the existing mobile payment system widely used in Europe which does not support this pattern, QR Code improves user experience and convenience.

As for financial security, both for individuals and enterprises, it would be better if the system could determine the user's real identity and limit each registered identity to only one individual account. Transactions which are illegal or suspected illegal can be easily tracked to a real identity.

To have effective trackability to the real identity, the system must have a way to identify users. Verification methods during registration are applied. Before submitting identity information, users must complete their email and mobile phone, which the system will automatically send verification code later. Only by completing the correct code can the user process. After submitting identity data, the system administrators must check the correctness and validity of the information submitted by the user, which includes passport number, passport photo, first name, last name, and nation. If the administrator rejects the registration, the registration will be cancelled. Otherwise, if he accepts, the user registration is successfully finished. Eventually, each user corresponds to one real identity. Users can sign up for merchants using the same account. The verification for merchants is similar. In verification, merchant name, merchant license number and merchant license photo must be provided, and the business entity must be the same as the user account's.

While a merchant account is bound with a user account, The system applies an independent pattern for merchant and user account, which means the income and outcome of one does not affect the other. Moreover, it is conducive to splitting different jobs in one account.

Furthermore, the system provides business APIs, including generating temporary payment links and refunding. These can be imported as Java libraries in their business systems.

1.4 Task Statement

1.4.1 System Description

The purpose of the automated mobile payment system is to facilitate the payment process between users and merchants and the transfer among users. In such a system, users can pay by scanning the QR code provided by the merchant, and they can also import and export their funds, check their bill records and register to be a merchant. For a merchant, the automated system provides them with various services, including setting QR code to receive payment, checking income and outcome, and embedding the payment system inside their system as a payment intermediate.

This project aims to create an android application called mobile payment, a web project for the merchant, and another web project for administrators and relevant databases.

In detail, there are three entrance clients in the system: one android application for users and merchants to operate basic operations, one web application for a merchant to embed the system ' s API, and the other for an administrator to manage the system.

1.4.2 Terminology in the System

- Session payment: a payment link that has temporary lifecycle and will expire after certain time.
- Payment server/Payment system: the system that the paper tries to implement.
- Merchant system/Merchant server: the merchant system running its own business. In this paper, it refers to the merchant system that calls the payment system's API.

1.4.3 Service Object

The system mainly serves for the following group of people:

- People in Belarus who like to use credit cards for shopping, and the system can replace their shopping habit of using credit cards.
- Small scale business merchants that rely on bank systems and POS machines. The system can offer them low rates of handling fees and save them the cost of purchasing payment terminals, like POS machines.

- Merchants that builds website and needs online payment. The system provides a Java library to provide access to the System's API and can be used to embed into their websites.

2 INFOMRATION SYSTEM STRUCTURE OF MOBILE PAYMENT SYSTEM

2.1 System Structure

2.1.1 IDEF0 diagram

IDEF0 has its beginning with Structured Analysis and Design (SADT). Softech, Inc. developed SADT in the mid−1970s in an effort to overcome some of the shortcomings of the modeling and analysis methods of that time. In the late 1970s, the Air Force selected SADT as the language to support its Integrated Computer Aided Manufacturing (ICAM) initiative. It was from this effort that IDEF0 was developed and brought into wide use, especially within the aerospace industry. Since that time much has been published in literature concerning the uses of IDEF0 and SADT. According to Ross, the methodology has been used by thousands of people from hundreds of organizations. Project areas have included system definition and design, project management and integration. Colquhounetal present an excellent review of applications to which IDEF0 have been applied. An organization for practitioners and researchers in IDEF methods, the IDEF Users Group, has been formed. This group holds regular meetings and conferences at which presentations and papers detailing research and case examples of IDEF uses are presented. The number of participants in these meetings and the number and quality of papers presented at the conferences are anecdotal evidence of the widespread adoption of the IDEF method [6].

There are five elements to the IDEF0 functional model (see figure 2.1): the activity (or process) is represented by boxes; inputs are represented by the arrows flowing into the left hand side of an activity box; outputs are represented by arrows flowing out the right hand side of an activity box; the arrows flowing into the top portion of the box represent constraints or controls on the activities; and the final element represented by arrows flowing into the bottom of the activity box are the mechanisms that carries out the activity. The inputs, control, output and mechanism arrows are also defined as ICOM's [6].

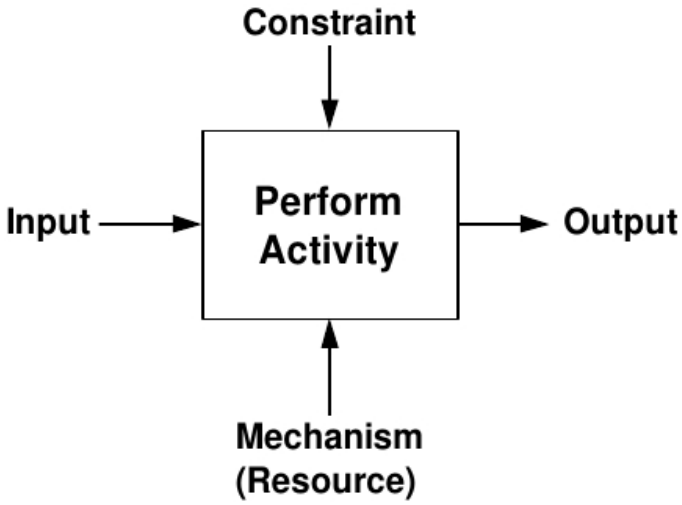


Figure 2.1 – IDEF0 Representation

Another characteristic of the IDEF0 modeling technique is that each activity and the ICOM's can be decomposed (or exploded) into more detailed levels of analysis. This is seen in Figure 2.2 below [6].

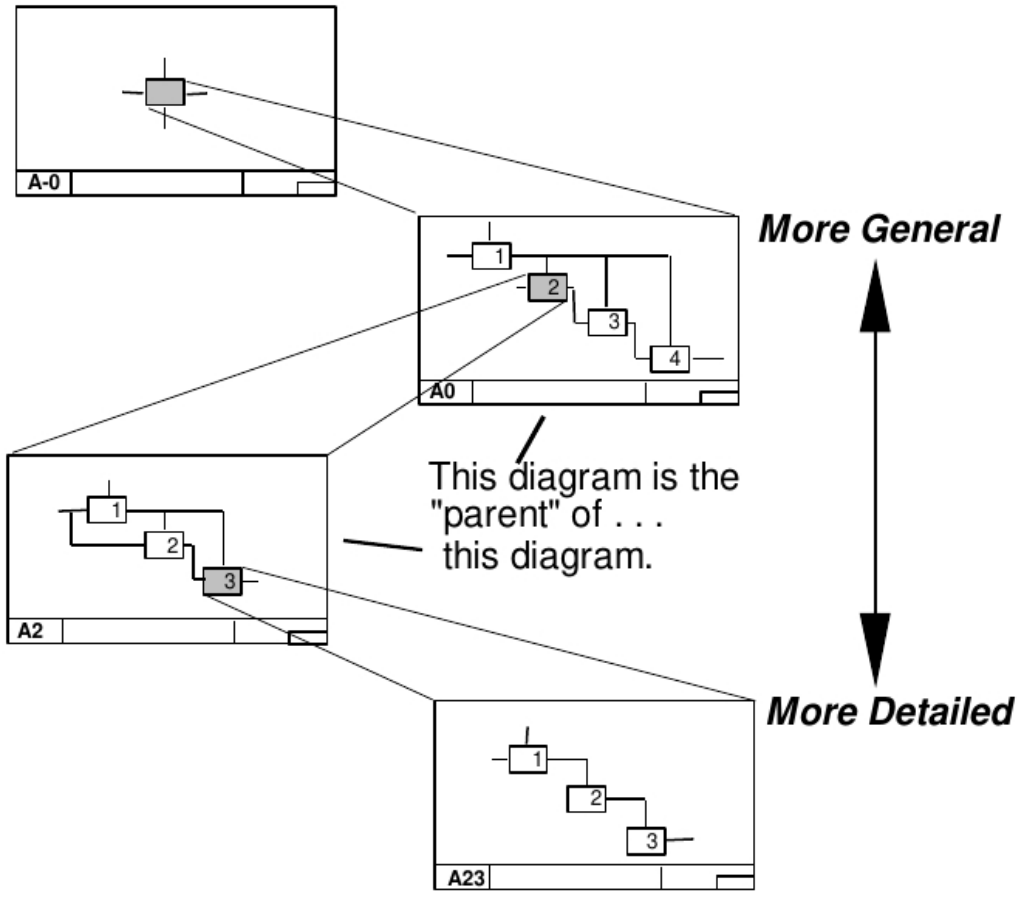


Figure 2.2 – Decomposition Overview

The following diagram (figure 2.3) depicts the IDEF0 diagram of the system. In the diagram, the input data is user authentication information, merchant identify and money amount. The system outputs hint message and payment information. Moreover, the constraints are HTTP Server, Database and Android device. User is the only mechanism (resource) of the system and the only executor of the mobile payment.

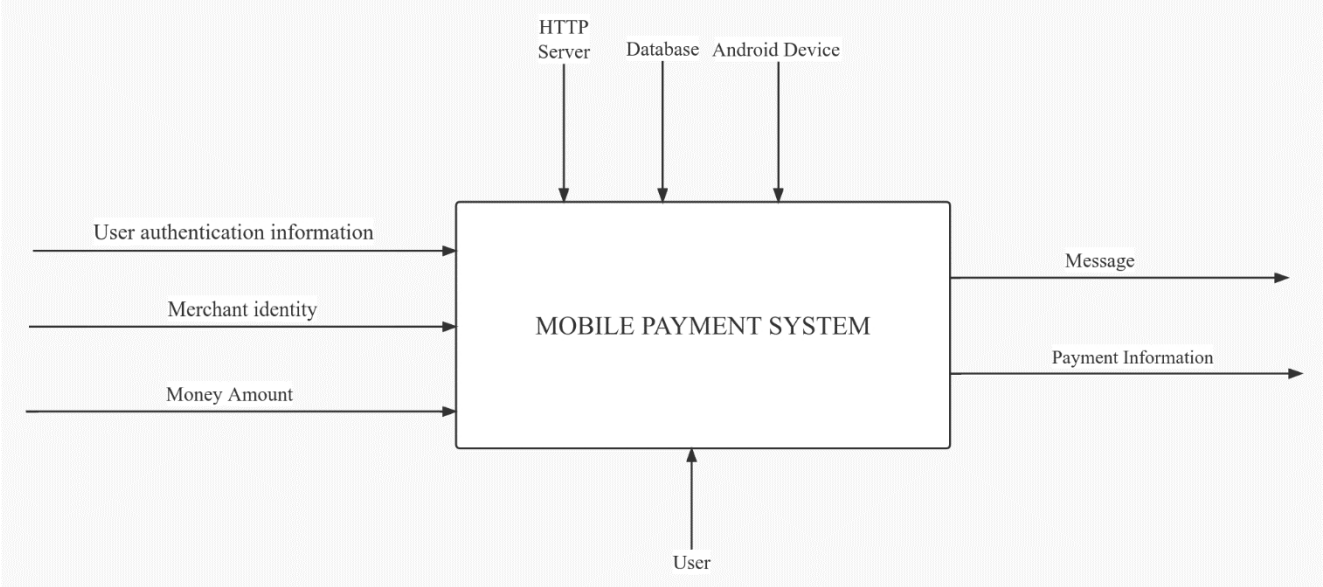


Figure 2.3 – Context diagram for mobile payment system

All the sub components of the IDEF0 diagram above are shown below (figure2.4).

In the payment function, there are seven sub – processes: Login, Scan merchant information, Add merchant balance, generate payment record, response, and rollback. Firstly, the user needs to login by providing certain authentication information. If the authentication is incorrect, a message indicating an invalid user is sent. Then the process scanning merchant information is needed. 'Invalid merchant message' is sent if the merchant identity cannot be recognized. Afterwards, it comes to three transactional processes – remove user balance, add merchant balance, and generate payment record. If any of the above three processes fails, it will call the rollback process and send an 'error' message – otherwise, it responds with successful payment information.

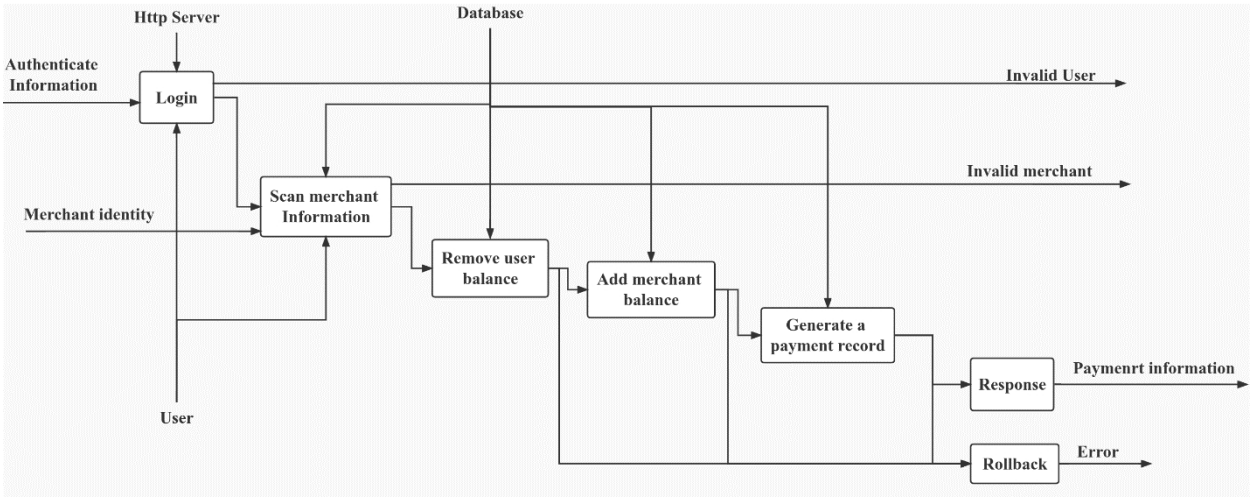


Figure 2.4 – Decomposition diagram of the System

2.2 UML Diagrams

Unified Modelling Language is a specification language that is used in the software engineering field. It can be defined as a general purpose language that uses a graphical designation which can create an abstract model. This abstract model can then be used in a system. This system is called the UML model. The Unified Modelling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modelling and other non−software systems[9].

2.2.1 Use Case diagram

A use diagram is a graphical depiction of possible interactions of different roles in the system. Typically, a use diagram can various use cases and different types of roles the system has. Three elements form use case diagram – Actors, Case and Relationship. The Actors are roles of the system, Case refers to the action or services the system can provide, and Relationships can be used to link between Actors and Case to form interactions or between Cases.

In this project, there are three Actors, which are Administrator, User and Merchant (figure 2.5).

The User can register, apply to be a merchant, import fund, export fund, transfer fund, make a pay, check balance, check bill and receive fund.

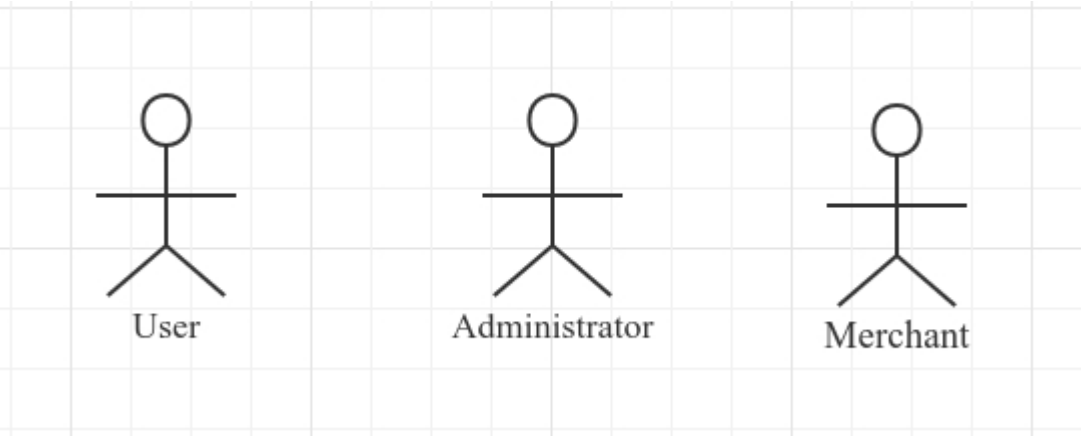


Figure 2.5 – Actors of the System

The Administrator is the role for management in the system. He/She can freeze user account, freeze merchant account, freeze user balance, freeze merchant balance, verify user registration by either rejecting or accepting and verify merchant application either by rejecting or accepting.

A Merchant evolves from User after the user ' s application and the administrator ' s acceptance. He/She can export funds, check balance, check bill, receive fund and call the system ' s payment API. The diagrams below (figure 2.6 and figure 2.7) are the corresponding use case diagrams.

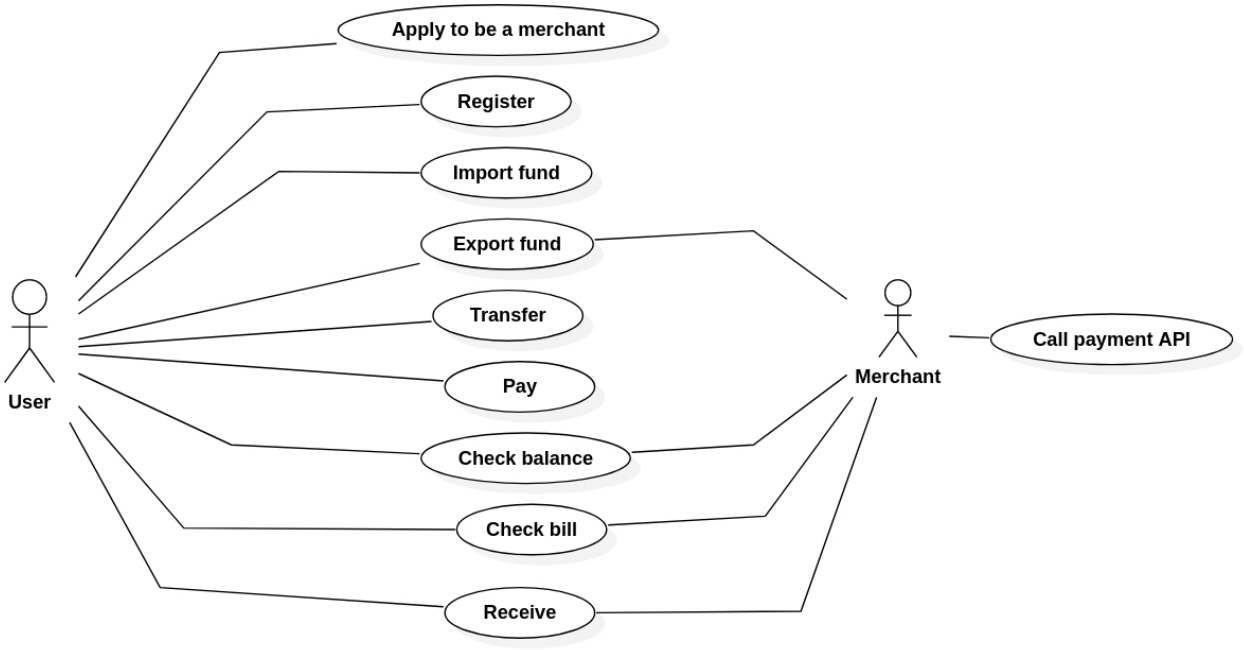


Figure 2.6 – User case diagram for User and Merchant

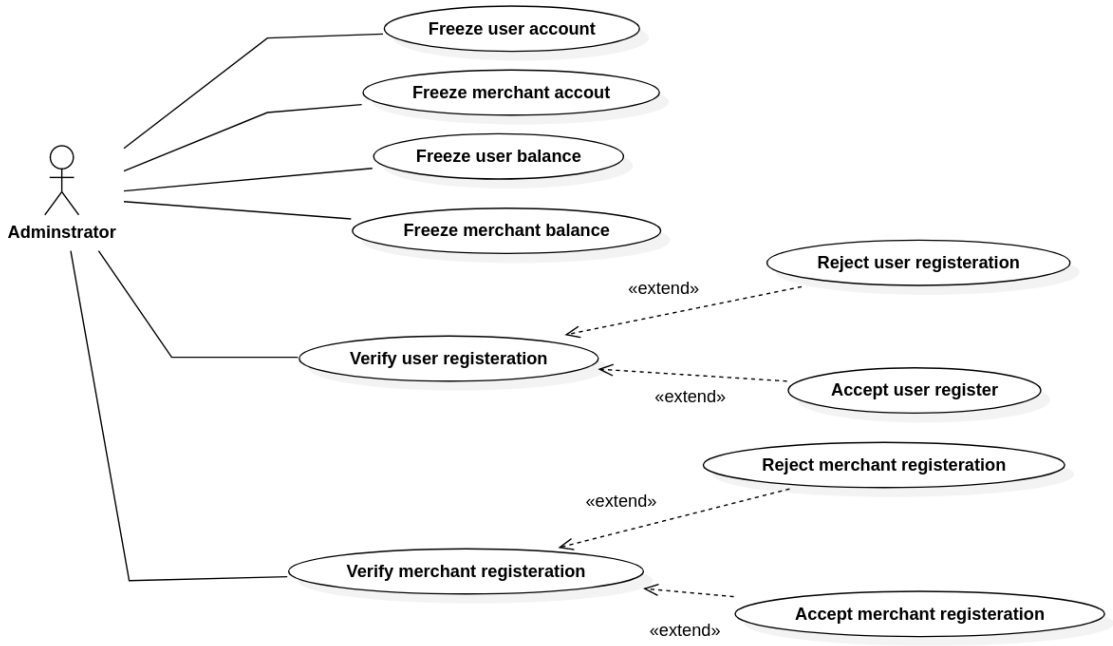


Figure 2.7 – User case diagram for Administrator

2.2.2 Sequence diagram

The popularity of the sequence diagram, originally called an object interaction diagram, is attributed to Jacobson et al. A sequence diagram focuses on time sequencing or time ordering of messages or the order in which messages are sent. The emphasis in these diagrams is what happens first, second, and so on. They represent the passage of time graphically. These diagrams have two axes: the horizontal axis displays the objects and the vertical axis shows time. In addition, sequence diagrams have two features not present in collaboration diagrams: an object's lifeline and the focus of control. Object lifelines are used in the sequence diagram to represent the existence of the object during a scenario. While most objects will be in existence during the entire scenario, at times objects are created or deleted during the scenarios. [10]

The following sequence diagram (figure 2.8) shows the interaction of session payment among the android app, browser, merchant server and mobile payment server. The sequence consist of following procedures:

1. The user logins to the system and fetches the mobile payment system token.
2. The user uses the browser to place an order at a merchant's website. On receiving user requests in the merchant's server, the merchant system sends a request for session payment to the mobile payment server. If successful, the payment server will return a payment link, and the merchant server will transfer the payment link back to the browser. On loading the payment link, the browser automatically starts a WebSocket with the mobile payment system.

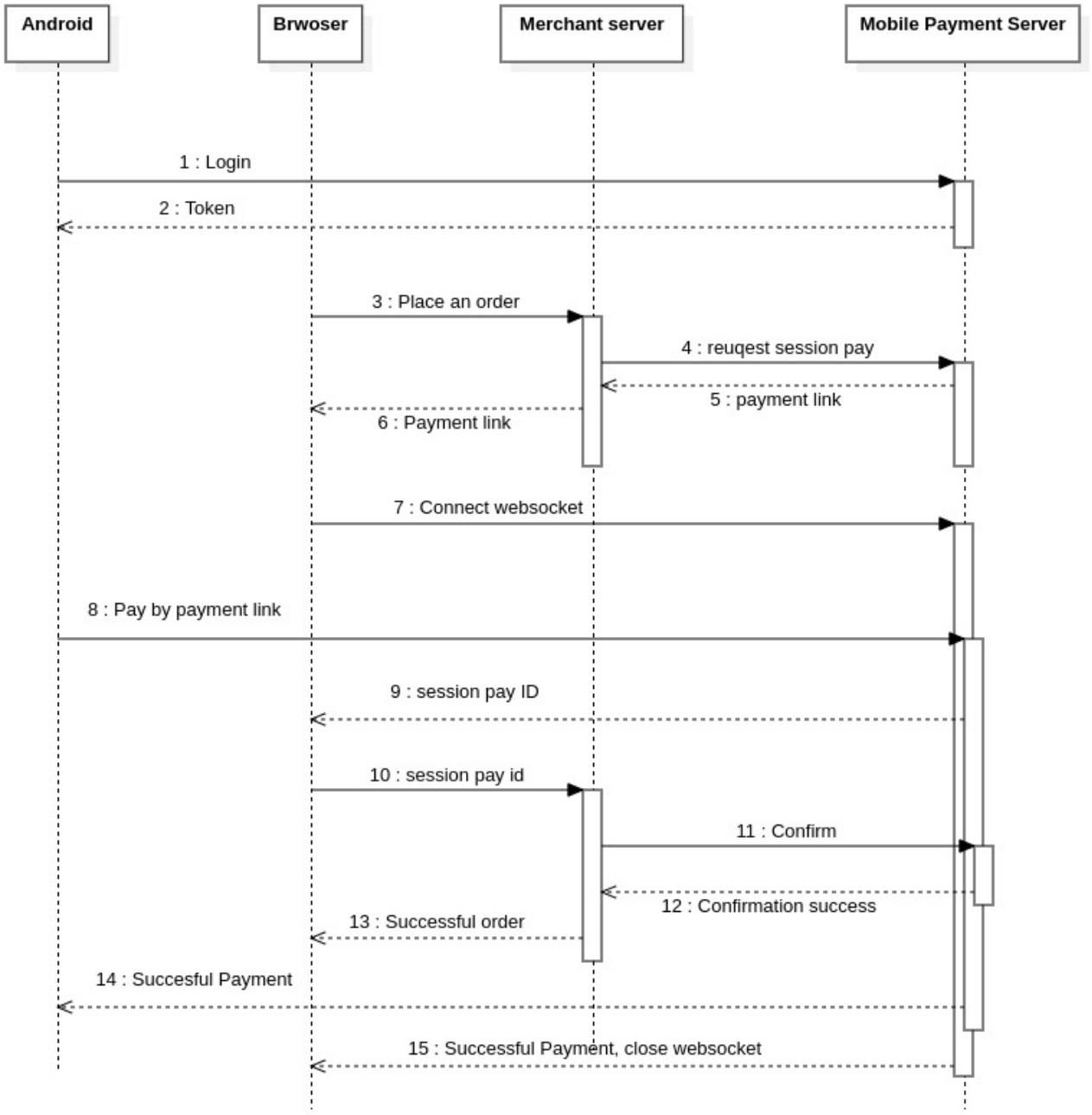


Figure 2.8 – Sequence diagram for session payment

3. The user uses the android app to resolve the payment link and pay. After clicking pay, the loading progress bar shows.
4. As the user pays, the WebSocket at the browser returns a session payment id which is used to send to the merchant server for verification.
5. The merchant sends a request for the session payment id verification and confirmation. If the verification is without error, the order placement is done.

6. Finally, the progress bar in procedure 3 ends with successful payment. The WebSocket ends with successful payment too.

2.2.3 Collaboration diagram

A collaboration diagram shows an interaction organized around the objects in the interaction and their links to each other. Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. On the other hand, a collaboration diagram does not show time as a separate dimension, so sequence numbers determine the sequence of messages and the concurrent threads.[11]

The following collaboration diagram (figure 2.9) depicts interactions among objects for the payment process.

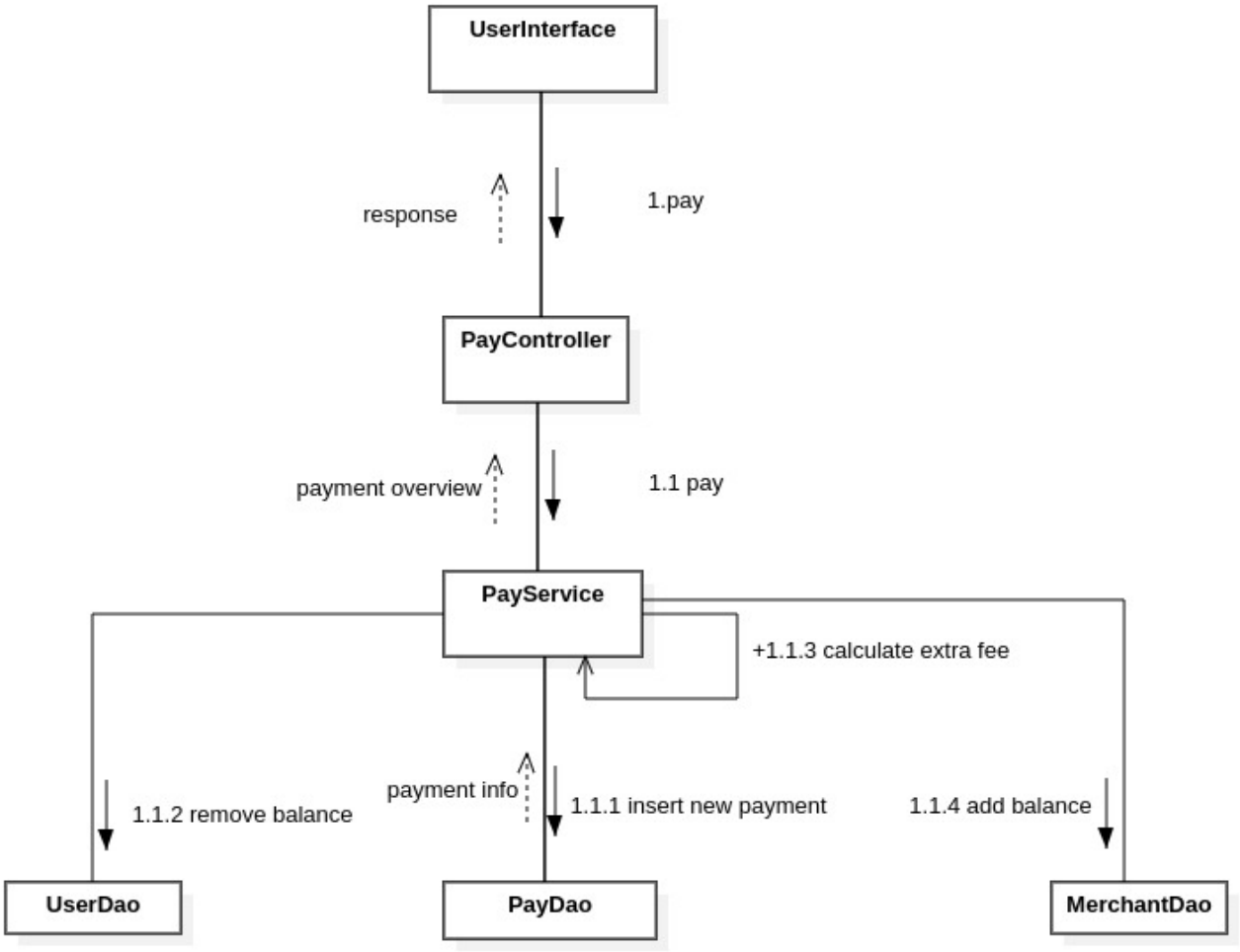


Figure 2.9 – Collaboration Diagram of Payment Process

In the process, when a user clicks pay on an android device, the class *UserInterface* automatically calls the pay method of *PayController*. Then inside the *PayController'* s method, it calls the pay method of *PayService*. During the

execution of the pay method of *PayService*, firstly it starts a transaction, secondly invokes the *PayDao*'s method to insert a new payment, thirdly calls *UserDao*'s method to update the user's balances, calculates the merchant's income in this payment and finally update merchant's balances. When all the steps above are finished, the *PayService* returns a payment overview to *PayController*, and *payController* will wrap the payment overview in the response.

2.2.4 Activity diagram

An activity diagram can be used to expand on a use – case description. Activity diagrams are similar to flow charts: they describe the order of activity and the branch logic of a process. However, they differ from traditional flow charts by allowing the representation of concurrent operations. Activities that take place simultaneously (such as threads) can be represented using activity diagrams. Activity diagrams can be used to supplement the use – case descriptions within a use–case model.[12]

Activity diagrams flow from top to bottom. The initial state is represented by a closed circle. Activity proceeds through a series of activity states until it reaches its final state, which is represented by a closed circle inside an open circle.Boxes with rounded corners represent activity states. Each activity state is labeled with a brief description of the activity it represents. The arrows between states, called transitions, represent the shift from one activity state to the next.[12]

The diagram below shows the processes of using the mobile payment system. When loading the page, a user enter his/ her credentials using the login column, then he/she is allowed to transfer, pay, check balance and check bills.

The activity below (figure 2.10) illustrates more detail.

For transfer operation, firstly, the system checks the validity of the source user and target user by searching the database. If the result turns out invalid, the operation fails. Otherwise, it then checks the state of both users on their account state to see if their account is unverified or frozen. Afterwards, if the source user has sufficient balance, the transfer operation will start.

For the payment operation, the procedures are similar. It checks the validity of the merchant and user and the user's balance. Then if the payment needs no confirmation, it starts pay operation. Otherwise, it awaits until a confirmation occurs in time. If no confirmation occurs in time, it will end up with failure.

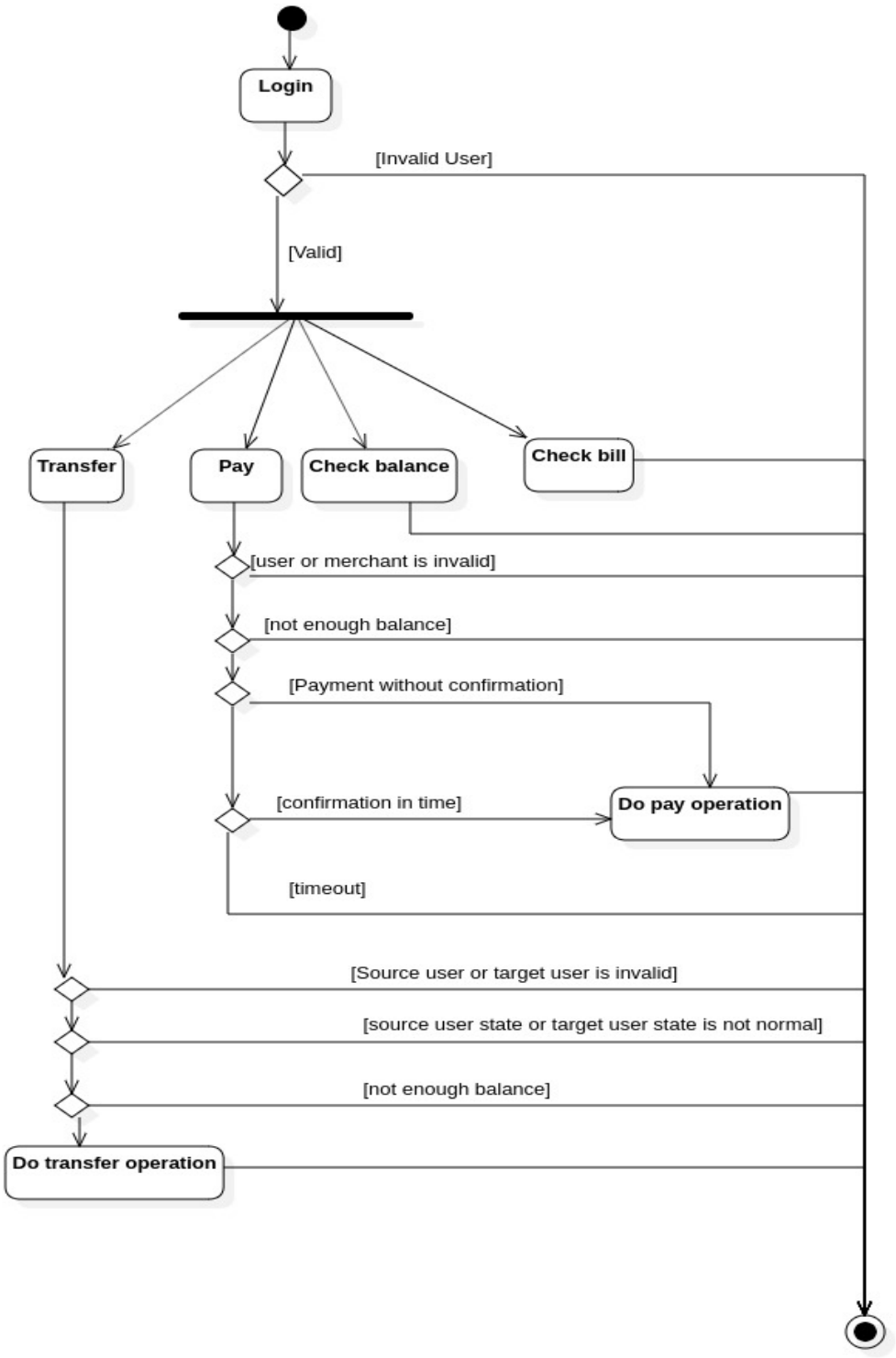


Figure 2.10 – Activity diagram

2.3 System Algorithm

2.3.1 Flowchart

A Flow Chart (also known as a Process Flow Diagram or Process Map) is a diagram of the steps in a process and their sequence. Two types of flow charts are utilized in quality improvement. A high – level flowchart, outlining 6 – 10 major

steps, gives a high – level view of a process. These flowcharts display the major blocks of activity, or the major system components, in a process. These charts are especially useful in the early phases of a project and help to set priorities for improvement work. A detailed flowchart is a close – up view of the process, typically showing dozens of steps. These flowcharts make it easy to identify complexity, excessive steps, etc. in a process and should be used when you want to standardize or make changes in the process. [7]

The following flowchart (figure 2.11) shows the process of session payment.

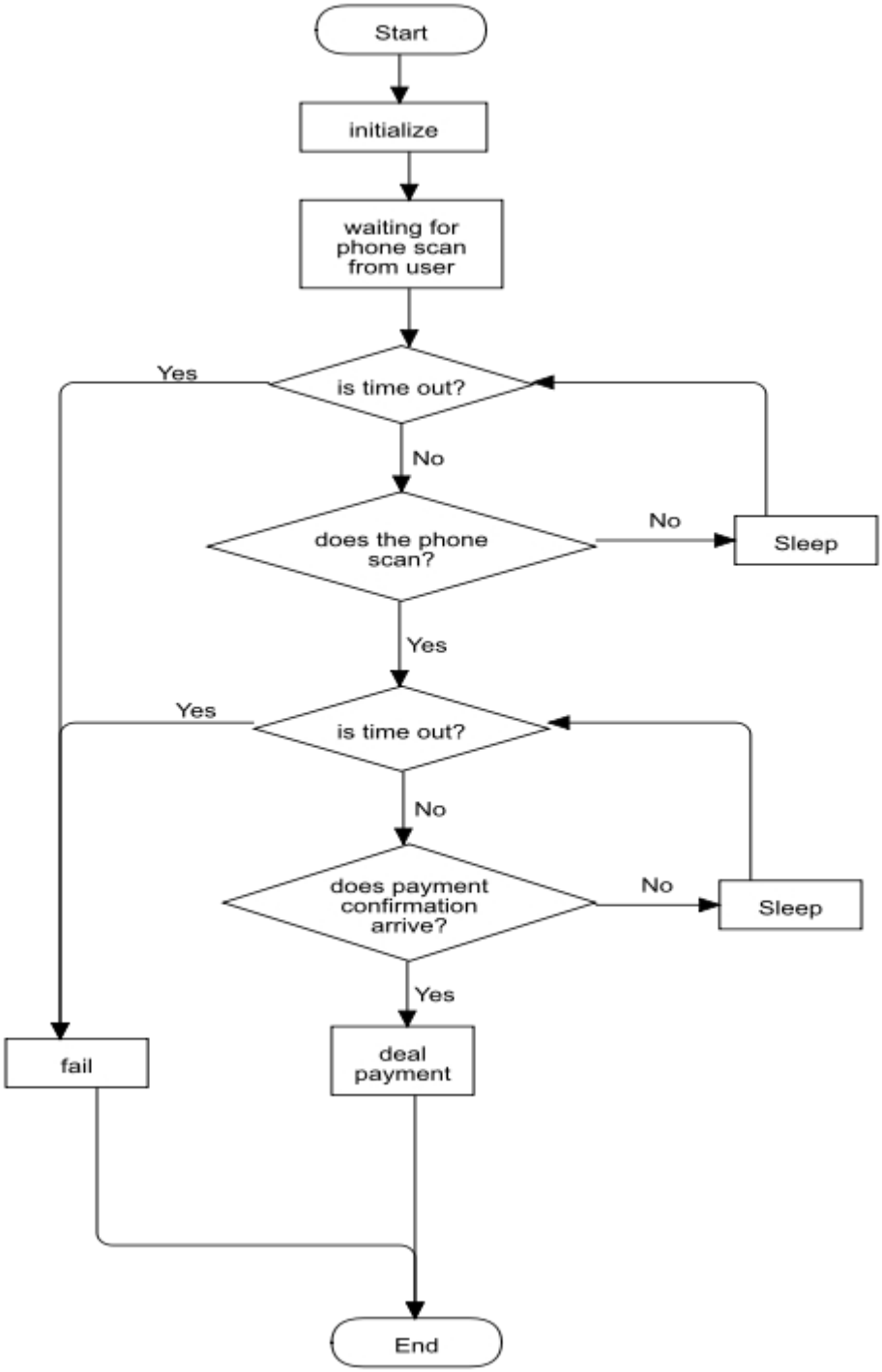


Figure 2.11 – Flowchart of session payment

Initially, the process begins by initializing some semaphore variables. Then it awaits until there occurs phone scan action. If the waiting time exceeds, the process ends up failing. Otherwise, if phone scans, it continues by waiting for confirmation from the merchant. If there is a merchant message during the expected time range, the process ends with success. Alternatively, it will fail due to timeout.

2.3.2 Algorithm Applied in Communication

2.3.2.1 Encryption Algorithm

The encryption algorithm applied in communication between merchant server and mobile payment server is RSA algorithm. The RSA algorithm is a very interesting cryptographic algorithm, and it is definitely one of the best and most secure algorithms available as of today. It provides great encryption and is reliable in terms of security and performance. The encryption security relies on the fact that the prime numbers used during the key generation process must be large enough to be unbreakable [8].

Before the communication between merchant servers and mobile payment servers begins, the merchant should register a 512 – bytes RSA public key and private, and payment system is supposed to store the merchant's public key. Also, the payment system should expose its public key to merchant server.

During the communication, all the content is encrypted with RSA algorithm. When the merchant server is ready to send plaintext, the plaintext is encrypted as cyphertext using the public key of payment system. When payment server receives cyphertext, it decrypts with its own private key. Then it checks the format of decrypted plaintext. If it fits certain JSON object format, then it will proceed. Otherwise, a wrong format error is responded.

2.3.2.2 Digital Signature Algorithm

The signature algorithm also applies RSA algorithm. Whenever content is ready from merchant server, an extra field called 'signature' is appended to the content to form plaintext. Then the plaintext is processed as described in the 'Encryption Algorithm' paragraph above. When the cyphertext is successfully decrypted in payment server, and the resulted plaintext has the right JSON format, then the payment server will check the content and signature, fetching merchant's public key and verifying it. If the verification result comes out successfully, it proceeds – otherwise, the 'wrong signature' error will be thrown.

The following steps illustrate the process above. Note that variable *C* stands for content, variable *S* stands for signature, and *E(X, K)* is a function that receives *X* and *K* and outputs the text that is RSA encryption of *X* with Key *K* . *D(X, K)* receives *X* and *K*, and outputs the text that is RSA decryption of *X* with Key *K*.

Procedure in merchant server:

Step 1: Start

Step 2: Input C

Step 3: S = D(C , merchant's private key)

Step 4: plaintext = {C , S}

Step 5: cyphertext = E(plaintext, payment server's public key)

Step 6: send cyphertext.

Procedure in payment server:

Step 1: Start

Step 2: Input cyphertext

Step 3: plaintext = D(cyphertext , payment server's private key)

Step 4: check whether plaintext fits the certain JSON format

Step 4.1 if plaintext does not fit the certain JSON format

Go to Step 9

Step 4.2 if plaintext fits the certain JSON format

Go to Step 5

Step 5: S = plaintext.signature, C = plaintext.content

Step 6: S1 = E(S, merchant's public key)

Step 7: check whether S1 is equal to C

Step 7.1 if S1 is equal to C

Go to Step 8

Step 7.2 if S1 is not equal to C

Go to Step 9

Step 8: execute operations, exit

Step 9: error

2.4 Design and Description of the Database System

A database is a more complex object; it is a collection of interrelated stored data that serves the needs of multiple users within one or more organizations, that is, interrelated collections of many different types of tables [13]. The motivations for using databases rather than files include greater availability to a diverse set of users, integration of data for easier access to and updating of complex transactions, and less redundancy of data.

Database design – is process of creating a database schema, and determining the necessary integrity constraints.

The main objectives of the database design are:

- to secure the database with all the necessary information;
- ensuring the possibility of obtaining all the necessary data requests;
- reducing redundancy and duplication of data;

- ensuring data integrity (correctness of their content): elimination of contradictions in the content of the data, with the exception of their loss. The database reflects information about a specific subject region.

A database schema represents the logical configuration of all or part of a relational database. It can exist both as a visual representation and as a set of formulas known as integrity constraints that govern a database. These formulas are expressed in a data definition language, such as SQL. As part of a data dictionary, a database schema indicates how the entities that make up the database relate to one another, including tables, views, stored procedures, and more[14].

Typically, a database designer creates a database schema to help programmers whose software will interact with the database. The process of creating a database schema is called data modelling. When following the three–schema approach to database design, this step would follow the creation of a conceptual schema. Conceptual schemas focus on an organization's informational needs rather than the structure of a database.

The following diagram shows the Entity Relationship of the database (figure 2.12).

For the system's ERD, there are 11 entities: Merchant, Export Record, RSA Key, Admin, User, Pay From To, Payment Order, Payment Refund, Import Record, Transfer From To and Transfer Record. Among these entities, entity RSA Key and Payment Refund are weak entities. Also, there are nine relationships in the ERD diagram. The following describes these relationships.

Export: the relationship between Export Record and Merchant, indicating that one merchant can have N export records.

Has Key: the relationship connecting entity Merchant and RSA Key and it means that each merchant can at most has one RSA public key.

Bind: the one–to–one relationship between Merchant and User, and the design satisfies the demand that each merchant account is registered by each user account.

Pay Flow: the ternary relationship among Merchant, User and Pay From To. It means that each user can pay each merchant many times, and each merchant can receive payment from each user many times.

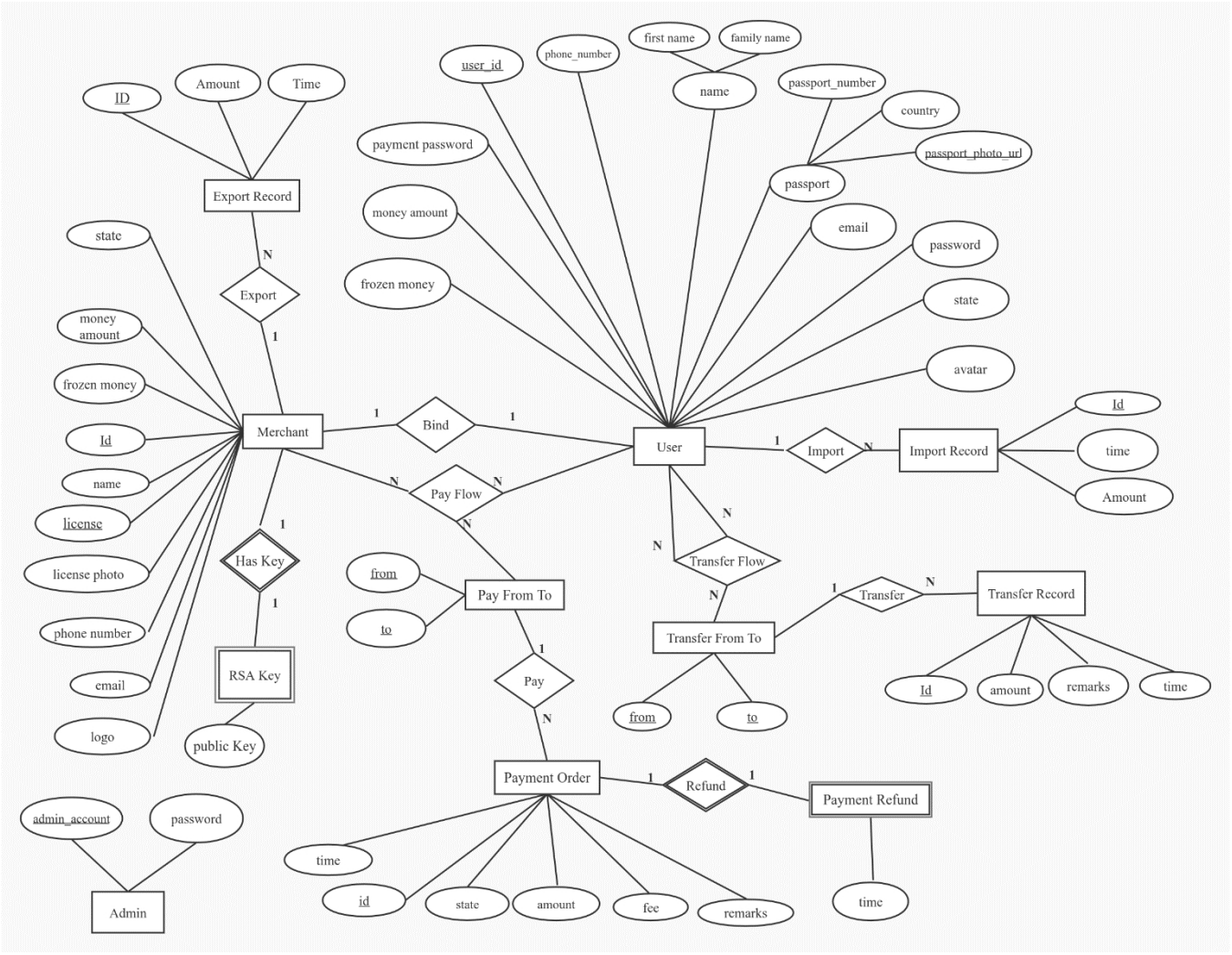


Figure 2.12 – Entity Relationship Diagram

Pay: the binary relationship between Payment Order and Pay From To. It shows that many payments can be made from the same user to the same merchant.

Import: the relationship between Export Record and Merchant, indicating that one user can have N import records.

Transfer Flow: the ternary relationship connecting user twice and Transfer From To once. It means that one user can transfer and receive funds many times.

Transfer: the relationship between Transfer From To and Transfer Record. It shows that many transfers can be made from the same source user to the same target user.

Refund: The identifying relationship between Payment Order and Payment Refund. Each payment order is bound with one payment refund and vice versa.

The following tables 2.1−2.11 are the detailed illustration of the ERD above.

Table 2.1 – Structure of *User* Entity

No.	Field name	Description
1	id	the identifier of user
2	phone number	user's phone number
3	first name	user's first name
4	last name	user's last name
5	passport number	user's passport number
6	country	user country
7	email	user email
8	passport photo	user's passport photo URL
9	password	user password
10	payment password	user password for payment
11	state	user state
12	money amount	user's balance
13	frozen money	the user's amount of frozen
14	avatar	the user's avatar

Table 2.2 – Structure of *Merchant* entity

No.	Field name	Description
1	id	the identifier of merchant
2	merchant name	merchant name
3	merchant license	merchant's license
4	merchant license_photo	merchant's license photo URL
5	merchant phone_number	merchant's phone number
6	merchant logo	merchant's logo
7	merchant email	merchant's email
8	frozen money	the merchant's amount of frozen money
9	money amount	the merchant's balance
10	state	the merchant state

Table 2.3 – Structure of *Admin* entity

No.	Field name	Description
1	admin account	the identifier of admin
2	password	admin password

Table 2.4 – Structure of *RSA Key* entity

No.	Field name	Description
1	public_key	merchant name

Table 2.5 – Structure of *Payment Order* entity

No.	Field name	Description
1	id	the identifier of payment
2	time	the payment time
3	state	payment state
4	amount	the amount involved in the payment
5	fee	the payment fee
6	remarks	the payment remarks

Table 2.6 – Structure of *Pay From To* entity

No.	Field name	Description
1	from	the payer of the payment
2	to	the payee of the payment

Table 2.7 – Structure of *Payment Refund* entity

No.	Field name	Description
1	time	the refund time

Table 2.8 – Structure of *Export Record* entity

No.	Field name	Description
1	id	the identifier of export
2	amount	the export amount
3	time	the export time

Table 2.9 – Structure of *Transfer Record* entity

No.	Field name	Description
1	id	the identifier of transfer
2	time	the transfer time
3	amount	the transfer amount
4	remarks	the transfer remarks

Table 2.10 – Structure of *Transfer From To* entity

No.	Field name	Description
1	from	the sender of the transfer
2	to	the receiver of the transfer

Table 2.11 – Structure of *Import Record* entity

No.	Field name	Description
1	id	the identifier of export
2	amount	the import amount
3	time	the import time

2.5 System Hardware and Software Requirements

Hardware and System Software requirements gives us the insight for developing the system and the minimum requirements needed to run the developed system.

Server Software Requirement:

- Operating System: Windows XP/Vista/2000/Windows 8/10, Linux;
- Presentation layer: CSS, HTML, JavaScript, VUE.
- Database: MYSQL;
- Runtime requirement: JDK 17.

Android Device Requirement:

- Operating System: Android with API >= 30.
- Android with back camera.
- Android with Internet access.
- Android with storage of more than 2.0G.
- Memory more than 1.5G.

Server Hardware Requirements:

The minimum requirements;

- Processor: Standard processor with a speed of 1.6 GHz;
- RAM:2 GB RAM or more;
- Hard Disk: 20 GB or more;

2.6 Ergonomics

The interaction of users with the systemwas carried out using Android and HTML. The interface of the system should be understandable and convenient, the system should not be overloaded with graphics and should provide a quick display of the screen. Navigation elements must be made in a user – friendly form. The means for updating information must satisfy the accepted agreements in terms of the use of function keys, operating modes, search, and use of the window system. The interface should correspond to modern Ergonomic requirements and provide easy access to the main functions and operations of the system.

The system should ensure correct handling of emergencies caused by incorrect user actions, invalid format or invalid input values. In these cases, the system must issue the appropriate messages to the user, and then return to the operational state that preceded the invalid (inadmissible) command or the incorrect

data entry. Screen forms should be designed taking into account the requirements of unification:

All the screen forms of the user interface must be executed in a single graphic design, with the same arrangement of the main controls and navigation;similar symbols, buttons and other control (navigation) elements should be used to indicate similar operations. The terms used to denote typical operations (adding an information entity, editing the data field), as well as the sequence of user actions when executing them, must be unified;

The external behavior of similar interface elements should be implemented identically for the same type of elements. The system must meet the requirements of ergonomics provided that it is equipped with high−quality equipment .

3 INFORMATION SYSTEM SOFTWARE IMPLEMENTATION

3.1 Software implementation Programming Tools

After considering the system's subject area and its structure, the most convenient way to implement the system is to use Kotlin for android apps in this system, Java for back–end development, JavaScript and VUE framework for front–end, Redis for cache and MYSQL for database. The following paragraph explains why.

3.1.1 Choosing Kotlin for Android Apps

Kotlin is a modern but already mature programming language aimed to make developers happier. It's concise, safe, interoperable with Java and other languages, and provides many ways to reuse code between multiple platforms for productive programming.[16] At Google I/O 2019, we announced that Android development will be increasingly Kotlin – first, and we ' ve stood by that commitment.[17]

Kotlin has been announced to be the first language of the android by Google, it has many new features compared to Java and it's developer friendly since it is more smart when dealing with nullable variables on android. So it's chosen.

3.1.2 Choosing Java for Back–End Development

Java is absolutely the most mature programming languages, and also it is object–oriented and support many object–oriented features, which means it can easy handle with situations where there is full of complex business logic. Also, with there is a powerful and easy–to–use web back–end framework 'Spring Boot' in Java. Similarly, other popular programming languages like C# and Python have the same features. However, since I'm to deploy my projects on Linux and C# is entirely not compatible with Non Windows OS, C# is not the choice. For Python, though it's one of most powerful languages, it has the shorting coming – it doesn't support static and inference types, thus it's hard to debug. It's not on the list neither.

3.1.3 Choosing JavaScript and VUE for Front–End

JavaScript, also named ECMAScript, is the first language in web, where it is the dominant language and supported by almost every browser. So, JavaScript is the best choice for front–end developing. For front–end framework, VUE is chosen. VUE is a JavaScript library for building user interface. It can provide HTML templates and bind JavaScript variables to the HTML UI's properties, providing MMVM patterns to separate program logic and user interface controls. Also, there

are many third–party libraries based on VUE, like Element UI and BootstrapVue. So, VUE is the on the list.

3.1.4 Choosing MYSQL for Database

The reason why choosing MYSQL is that MYSQL is an open – source relational database system. It supports multi – platforms and most functions can perform well on Linux. Compared to other popular DBMS, like Oracle and SQL Server, MYSQL is a free product, the developing cost is zero.

3.1.5 Choosing Redis for Cache

Redis is an open source (BSD licensed), in–memory data structure store used as a database, cache, message broker, and streaming engine.[19] In Redis, data structures like strings, hashes, lists, sets and other common ones are provided. Another import feature this project preferred is the temporary data storage. Redis provides temporary storage, which users can set the data's expire time. Compared to storing it in traditional SQL database, where there must exist structural table and expired data can't be automatically removed, Redis supports KV storage. Furthermore, Redis is provided in memory, thus it's much faster to access.

3.2 Software Structure

The diagram below (figure 3.1) shows the system components.

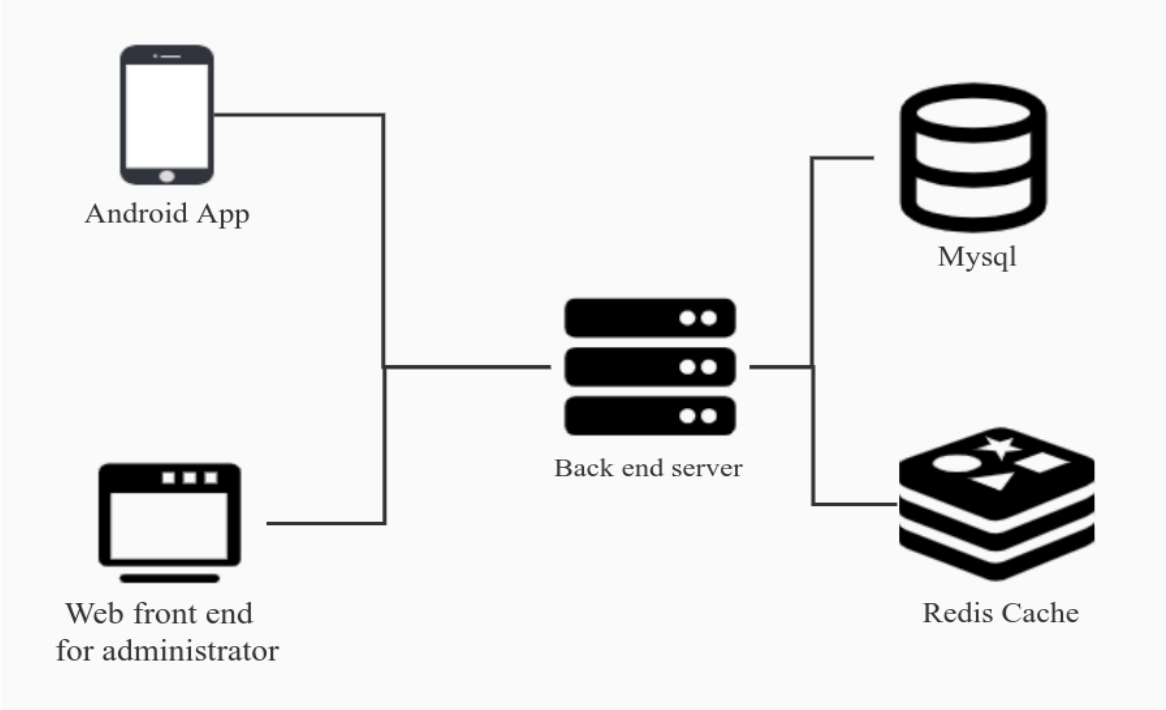


Figure 3.1 – System Component Diagram

The protocol used in communication between client and server is HTTPS, and the way Back–end server communicates with MySQL and Redis Cache is by

TCP protocol.Automated mobile payment system has five components, including one android app, one web front–end for administrator, one back–end server, one MySQL database and one Redis Cache.

3.2.1 Android App Software Structure

The Android App uses the latest MVVM pattern, which includes Model, View and View–Model. Additionally, all the data in View–Model is dependent on a persistent layer. The persistent layer could be A SQL database or a Key–Value database. Variables in View–Models can be LiveData whose value's change leads to UI(View)'s update, and view–model can start corountines, listening for data change in persistence layer. Whenever data in persistence layer is modified, the flow notifying change and carrying new data is received in View–Model, therefore the LiveData in View–Model is changed and UI is updated too. The following figure (figure 3.2) illustrates the structure.

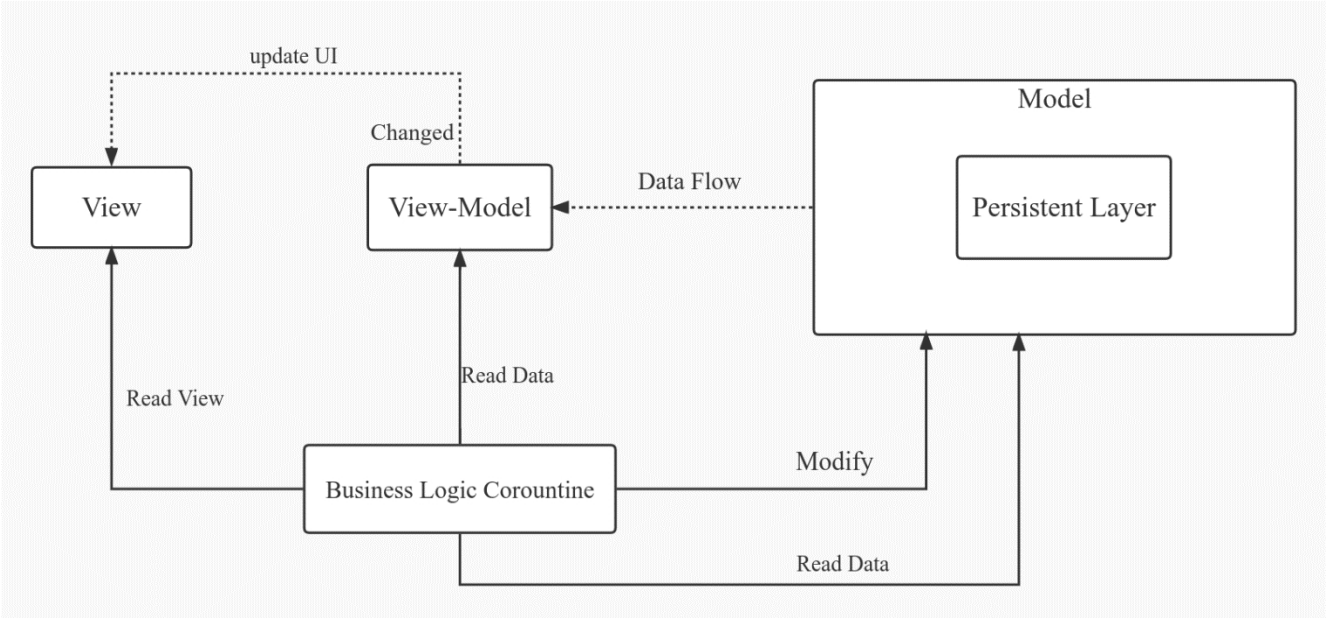


Figure 3.2 – Android App Structure

The Business Logic Corountine can read view from View, read data from View–Model and modify and read the data in persistent layer.

3.2.2 Front–End Software Structure

Similarly, the Front–End's structure applies the same pattern, MVVM, except that the View is HTML, View–Model is coded in JavaScript and Model doesn't contain Persistent layer. Also, Business Logic Corountine is changed to be Business Logic Functions.

3.2.3 Back–End Software Structure

The following diagram (figure 3.3) shows the back–end system structure.

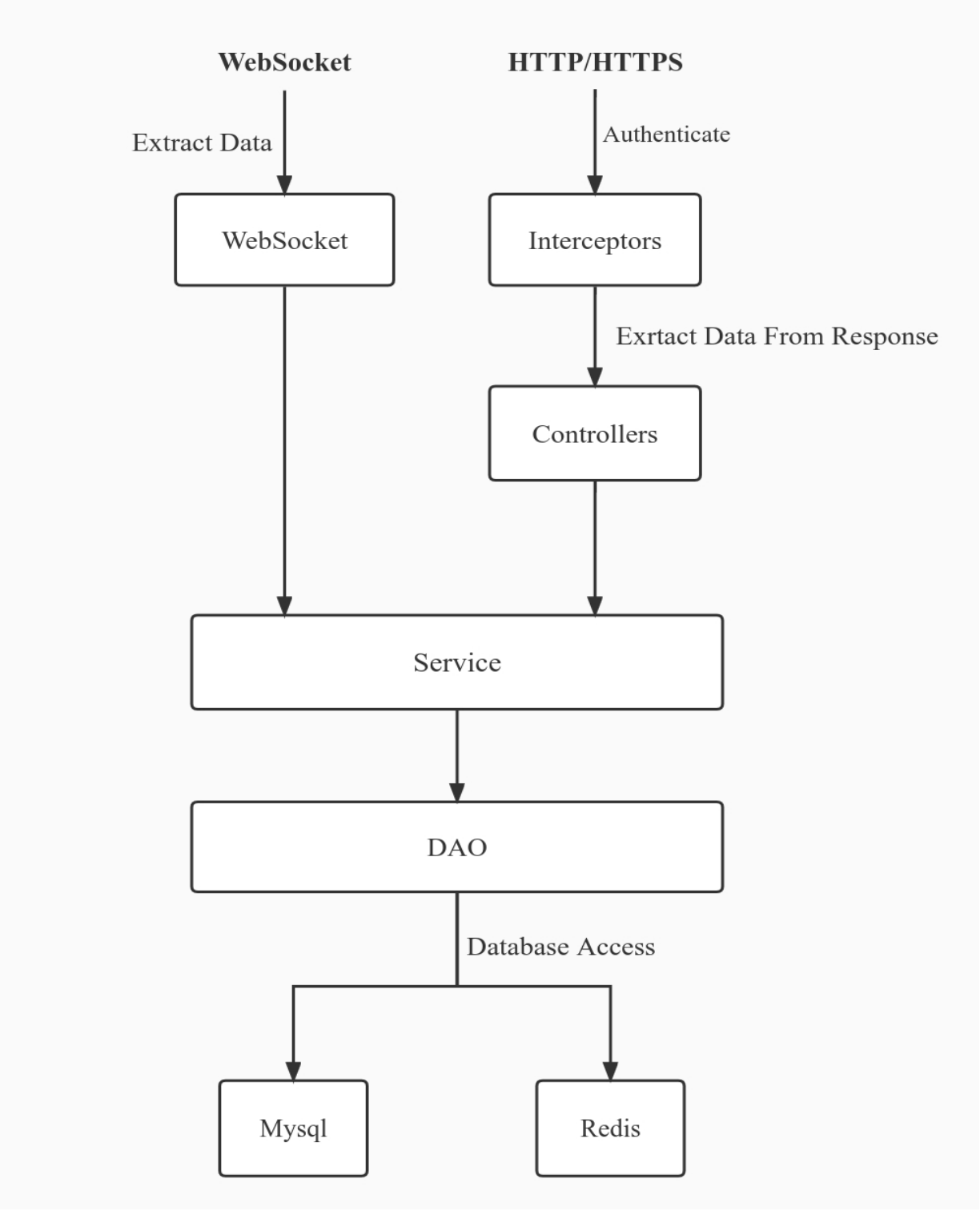


Figure 3.3 – Back–End System Structure

The framework used in back–end development is Spring Boot. In this project, there are two types of entry points, HTTP request and WebSocket. Before HTTP request arriving in Controllers, it is checked by Interceptors, which authenticates the HTTP request and accept or reject the request based on the token carried. The

Controllers' job is to extract data from request and wrap data to response and the extracted data is dealt in Service layer. The Service layer mainly involves with business logic, like checking, verifying and so on. Also, it invokes the methods provided by DAO layer to persistent data. The DAO is short of Data Access Object whose job is provide access to database. For WebSocket, when dealing with business logic, it also extracts corresponding data and pass it to Service layer.

3.2.4 The database structure

The following figure is the the database's physical model. It is based on MYSQL database at version 5.7 (figure 3.4) .

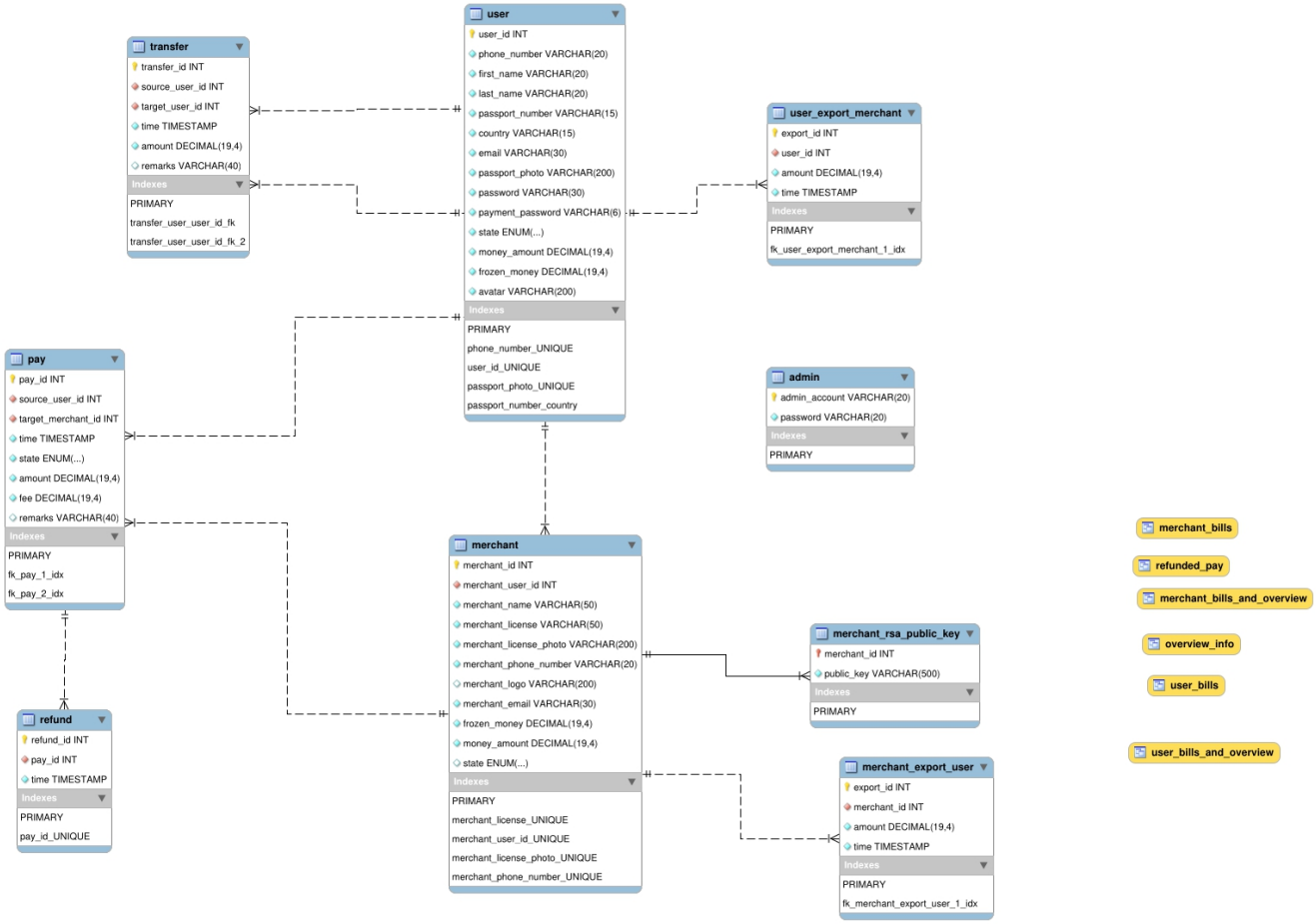


Figure 3.4 – Physical Model

The following tables are detailed structure of the physical model.

Table 3.1 – Structure of *user* table

No.	Field name	Datatype	Description
1	user_id	INT	the identifier of user
2	phone_number	VARCHAR(20)	user's phone number
3	first_name	VARCHAR(20)	user's first name
4	last_name	VARCHAR(20)	user's last name
5	passport_number	VARCHAR(20)	user's passport number
6	country	VARCHAR(15)	user country
7	email	VARCHAR(30)	user email
8	passport_photo	VARCHAR(200)	user's passport photo URL
9	password	VARCHAR(30)	user password
10	payment_password	VARCHAR(6)	user password for payment
11	state	ENUM(normal, frozen, unverified)	user state
12	money_amount	DECIMAL(19,4)	user's balance
13	frozen_money	DECIMAL(19,4)	the user's amount of frozen money
14	avatar	VARCHAR(200)	the user's avatar

Table 3.2 – Structure of *merchant* table

No.	Field name	Datatype	Description
1	merchant_id	INT	the identifier of merchant
2	merchant_name	VARCHAR(50)	merchant name
3	merchant_license	VARCHAR(50)	merchant's license
4	merchant_license_photo	VARCHAR(200)	merchant's license photo URL
5	merchant_phone_number	VARCHAR(20)	merchant's phone number
6	merchant_logo	VARCHAR(200)	merchant's logo
7	merchant_email	VARCHAR(20)	merchant's email
8	frozen_money	DECIMAL(19,4)	the merchant's amount of frozen money
9	money_amount	DECIMAL(19,4)	the merchant's balance
10	state	ENUM(normal, frozen, unverified)	the merchant state
11	merchant_user_id	INT	the user id the merchant bound to

Table 3.3 – Structure of *admin* table

No.	Field name	Datatype	Description
1	admin_account	VARCHAR(20)	the identifier of admin
2	password	VARCHAR(20)	admin password

Table 3.4 – Structure of *merchant_rsa_public_key* table

No.	Field name	Datatype	Description
1	merchant_id	INT	public key owner's id
2	public_key	VARCHAR(500)	merchant name

Table 3.5 – Structure of *pay* table

No.	Field name	Datatype	Description
1	pay_id	INT	the identifier of payment
2	source_user_id	INT	the user id of payer
3	target_merchant_id	INT	the merchant id of payee
4	time	TIMESTAMP	the payment time
5	state	ENUM(normal, frozen, unverified)	payment state
6	amount	DECIMAL(19,4)	the amount involved in the payment
7	fee	DECIMAL(19,4)	the payment fee
8	remarks	VARCHAR(40)	the payment remarks

Table 3.6 – Structure of *refund* table

No.	Field name	Datatype	Description
1	refund_id	INT	the identifier of refund
2	pay_id	INT	the identifier of payment the refund linked
3	time	TIMESTAMP	the refund time

Table 3.7 – Structure of *merchant_export_user* table

No.	Field name	Datatype	Description
1	export_id	INT	the identifier of export
2	merchant_id	INT	the identifier of merchant the refund linked to
3	amount	DECIMAL(19,4)	the export amount
4	time	TIMESTAMP	the export time

Table 3.8 – Structure of *transfer* table

No.	Field name	Datatype	Description
1	transfer_id	INT	the identifier of transfer
2	source_user_id	INT	the user id of transfer source
3	target_user_id	INT	the user id of transfer target
4	time	TIMESTAMP	the transfer time
5	amount	DECIMAL(19,4)	the transfer amount
6	remarks	VARCHAR(40)	the transfer remarks

Table 3.9 – Structure of *user_export_merchant* table

No.	Field name	Datatype	Description
1	export_id	INT	the identifier of export
2	user_id	INT	the id of the user who export funds
3	amount	DECIMAL(19,4)	the export amount
4	time	TIMESTAMP	the exporting time

3.2.5 Package Specification

The Android app consists of following several packages:

- *entity*: the package containing the class of entity class, including the data transfer object.
- *network*: the package containing class of network configure and classes that request system rest API.
- *room.roomEntity*: the package involving class to be persistent using room framework
- *room.roomDao*: the package involving ORM classes related with entities in package room.roomEntity and package entity.
- *ui.lib:* library that can be reused.
- *ui.login*: the packages containing UI and business logic classes related with login page.
- *ui.mainPage*: the packages containing UI and business logic classes related with main page.
- *ui.merchantRegister*: the packages containing UI and business logic classes related with merchant register page.
- *ui.register*: the packages containing UI and business logic classes related with user register page.

The back–end consists of following several packages:

- *businessEntity*: the package containing entity used in business logic code
- *config*: The Spring Boot configure package, including static file configure, WebSocket configure, Swagger configure and other common configure.
- *controller*: The package containing controller classes.
- *dao*: The package containing data access object.
- *dto*: Data transfer object package.
- *entity*: Entity package
- *enums*: The package containing *enums*.
- *interceptors*: The package containing classes related with interceptors.
- *redisDao*: The package containing Redis DAO.
- *redisEntity*: The entity that persistent in Redis.
- *service*: The package containing service classes.
- *util*: The util class package.

- *websocket*: The WebSocket class package.

3.3 Code Description

The following code is related with payment (not session payment)

The function pay receives *userId*, *merchantId*, amount, *paymentPassword* and *remarks*. It tries to generate a payment of certain amount between a user and a merchant with payment password and remarks, and it outputs the payment response.

```
@ApiImplicitParams({
    @ApiImplicitParam(name = "token", paramType = "header"),
})
@PostMapping("/api/pay")
public ResponseData<PayResp> pay(@ApiIgnore @RequestAttribute("userId") int userId,
                    @RequestParam(name = "merchantId") int merchantId,
                    @RequestParam(name = "amount") BigDecimal amount,
                    @RequestParam(name = "paymentPassword") String paymentPassword,
                    @RequestParam(name = "remarks")String remarks
                    ) {

  amount = amount.setScale(4, RoundingMode.HALF_UP);
  ResponseData<PayResp> responseData = new ResponseData<>();
  responseData.data = new PayResp();
  try{
    var promptAndPay = payService.pay(userId, merchantId, amount, paymentPassword,
remarks);
    responseData.data.prompt = (Prompt) promptAndPay[0];
    responseData.data.payOverview = PayOverview.fromPay((Pay) promptAndPay[1]);
  } catch (Exception e) {
    e.printStackTrace();
    responseData.errorPrompt = "Error";
    responseData.status = ResponseData.ERROR;    }
  return responseData;}
```

In service layer, the method receives *userId*, *merchantId*, *amount*, *paymentPassword* and *remarks*. It tries to generate a payment order by invoking _pay method. If the inserting process fails, it will return fail status – otherwise, it will return the generated payment record.

```
public Object[] pay(int userId, int merchantId, BigDecimal amount, String paymentPassword,
String remarks) {
  Prompt prompt = Prompt.pay_error;
  Prompt[] returnedPrompt = new Prompt[]{prompt};
```

```
Pay pay = null;
try {
    pay = transactionHandler.runInTransactionSerially(() -> _pay(userId, merchantId,
amount, paymentPassword, remarks, returnedPrompt));
} catch (Exception e) {
    e.printStackTrace();    }
prompt = returnedPrompt[0];
return new Object[]{prompt, pay};}
```

This methods tries to generate a payment record with given parameters. In the function, transaction is made. If the manipulation fails, null is returned. Or generated record is returned.

```
private Pay _pay(int userId, int merchantId, BigDecimal amount, String paymentPassword,
String remarks,
            Prompt[] returnedPrompt) {
    User user = userDao.selectById(userId);
    Merchant merchant = merchantDao.selectById(merchantId);
    LambdaLogicChain<Prompt> logicChain = new LambdaLogicChain<>();
    returnedPrompt[0] = logicChain.process(
() -> user == null ? Prompt.pay_user_not_found_error : null,
() -> merchant == null ? Prompt.pay_merchant_not_found_error : null,
()->!user.paymentPassword.equals(paymentPassword) ?
        Prompt.payment_password_not_correct : null,
() -> amount.compareTo(BigDecimal.ZERO) <= 0 ? Prompt.pay_amount_invalid_error : null,
() -> user.state == State.frozen ? Prompt.pay_user_account_frozen : null,
() -> user.state == State.unverified ? Prompt.pay_user_account_unverified : null,
() -> user.userId.equals(merchant.merchantUserId) ? Prompt.pay_user_to_self_error : null,
() -> merchant.state == State.unverified ? Prompt.pay_merchant_account_unverified : null,
() -> merchant.state == State.frozen ? Prompt.pay_merchant_account_frozen : null,
() -> user.moneyAmount.compareTo(amount) < 0 ? Prompt.pay_user_not_enough_balance :
null,
 () -> Prompt.pay_error);
    if (returnedPrompt[0] != Prompt.pay_error) {
        throw new RuntimeException();
    }
```

The following code tries to remove balance from user and update the database.

```
  user.moneyAmount = user.moneyAmount.subtract(amount).setScale(4,
RoundingMode.HALF_UP);
    userDao.updateById(user);
```

The following code tries to insert a payment record into database.

```
    Pay pay = new Pay();
    pay.amount = amount;
    pay.fee = amount.multiply(ConfigUtil.FEE_RATE).setScale(4, RoundingMode.HALF_UP);
    pay.sourceUserId = userId;
    pay.targetMerchantId = merchantId;
    pay.remarks = remarks;

    payDao.insert(pay);
```

The following code tries to add the payment income to the merchant after deducting the fee.

```
        merchant.moneyAmount = merchant.moneyAmount.add(amount.subtract(pay.fee));
    merchantDao.updateById(merchant);

    returnedPrompt[0] = Prompt.success;

    return payDao.selectById(pay.payId);
}
```

The following code is related with session payment. This method receives encrypted message,decodes the message, verify signature, initializes variables for session payment and returns a response.

```
@PostMapping("/api/sessionPay")
public ResponseData<SessionPayResp> requestSessionPay(
@RequestBody String RSAEncryptedBase64String) {
   ResponseData<SessionPayResp> resp = new ResponseData<>();
   resp.data = new SessionPayResp();

    try {

        //check the request format
        var sessionRequest = sessionPayService.extractInfo(RSAEncryptedBase64String);
        if (sessionRequest == null) {
```

```
            resp.data.prompt = Prompt.session_pay_request_format_error;
            return resp;
        }

        //verify the signature
        Prompt prompt = sessionPayService.verifySessionRequest(sessionRequest);
        if (prompt != Prompt.success) {
            resp.data.prompt = prompt;
            return resp;
        }
```

The following code fragment initialize pay semaphore

```
        SessionPay sessionPay = sessionPayService.initialize(sessionRequest.merchantId,
sessionRequest.amount);

        //initialize pay semaphore
        PaySemaphore paySemaphore = new PaySemaphore(sessionPay.sessionId);
        PaySemaphorePool.getInstance().add(paySemaphore);


        resp.data.sessionId = sessionPay.sessionId;
        resp.data.prompt = Prompt.success;

    } catch (Exception e) {
        e.printStackTrace();
        resp.data.prompt = Prompt.unknownError;
    }

    return resp;
}
```

This method provides rest API for phone scan. While scanning the session pay QR Code, phone will pass authentication data which will be userId later and other relevant parameters. This method will notify the WebSocket to send phone paid message by releasing the semaphore. The android app requesting this API will be loading until the session payment is confirmed or timeout.

```
@PostMapping("/api/payWithConfirm")
public ResponseData<PayResp> payWithConfirm(@ApiIgnore @RequestAttribute("userId") int
userId,
        @RequestParam(name = "sessionId") int sessionId,
        @RequestParam(name = "paymentPassword") String paymentPassword,
        @RequestParam(name = "remarks") String remarks) {
```

```
ResponseData<PayResp> responseData = new ResponseData<>();
responseData.data = new PayResp();

try {

   //if the it is not initialized
   PaySemaphore paySemaphore = PaySemaphorePool.getInstance().get(sessionId);
   if (paySemaphore == null) {
      responseData.data.prompt = Prompt.pay_session_id_error;
      return responseData;
   }
```

The following code block guarantees only one mobile phone can scan at one time. If it is already scanned by other phone, the method will return *multiple user pay error* immediately.

```
   try {
      var isOkay = paySemaphore.notScanned.tryAcquire(10, TimeUnit.SECONDS);
      if (!isOkay)
         throw new InterruptedException();
   } catch (InterruptedException e) {
      e.printStackTrace();
      responseData.data.prompt = Prompt.multiple_user_pay_error;
      return responseData;
   }
```

The following codes try to make session payment ready to be paid. If it successfully modifies the session payment's state, it will notify other thread waiting for the semaphore.

```
      Prompt prompt = sessionPayService.phoneScan(sessionId, userId, remarks,
paymentPassword);
      if (prompt != Prompt.success) {
         paySemaphore.notScanned.release();
         responseData.data.prompt = prompt;
         return responseData;
      }

      //notify other thread that waiting for user payment
      paySemaphore.isPaid.release();

      //check whether payment  is finished
      try {
```

```
            var isOkay = paySemaphore.isFinished.tryAcquire(1, TimeUnit.MINUTES);
            if (!isOkay)
                throw new InterruptedException();
        } catch (InterruptedException e) {
            e.printStackTrace();

            //reset back
            paySemaphore.isPaid.acquire();

            //reset  scan
            paySemaphore.notScanned.release();
            responseData.data.prompt = Prompt.pay_time_out;
            return responseData;
        }
```

The following code blocks try to do the final jobs, release *isFinish*, reset *isPaid* and *phoneScan.*

```
            paySemaphore.isFinished.release();

            //reset back
            paySemaphore.isPaid.acquire();

            //reset phone scan
            paySemaphore.notScanned.release();

            responseData.data = paySemaphore.paySynData.payResp;

            return responseData;

        } catch (Exception e) {
            e.printStackTrace();
            responseData.data.prompt = Prompt.unknownError;
            return responseData;
        }
    }
```

This method is the function for merchant to confirm the session payment. It receives encrypted message as the parameter. It will first decrypt and check the message. If it goes well, then it will check the payment semaphore. Afterwards, it tries to persistent the payment into database. Finally, releasing certain semaphores and make scanning phone's loading state finished.

```
@PostMapping("/api/payVerify")
public ResponseData<VerifyPayResp> verifyPay(@RequestBody String
RSAEncryptedBase64String) {

    ResponseData<VerifyPayResp> resp = new ResponseData<>();
    resp.data = new VerifyPayResp();
    resp.data.prompt = Prompt.unknownError;

    try {

        //extract information
        MerchantVerifyInfo verifyInfo = payVerifyService.extractInfo(RSAEncryptedBase64String);
        if (verifyInfo == null) {
            resp.data.prompt = Prompt.pay_verify_request_format_error;
            return resp;
        }

        //fetch sessionPay and check whether it exists or not
        SessionPay sessionPay = sessionPayService.getById(verifyInfo.sessionId);
        if (sessionPay == null) {
            resp.data.prompt = Prompt.pay_time_out;
            return resp;
        }

        // get the paySemaphore and check whether it exists or not
        PaySemaphore paySemaphore =
PaySemaphorePool.getInstance().get(sessionPay.sessionId);
        if (paySemaphore == null || paySemaphore.notScanned.availablePermits() == 1) {
            resp.data.prompt = Prompt.pay_time_out;
            return resp;
        }

        //verify the signature
        Prompt prompt = payVerifyService.verify(verifyInfo, sessionPay);
        if (prompt != Prompt.success) {
            resp.data.prompt = prompt;
            return resp;
        }
```

To start persistent payment into database we use the following code:

```
        Object[] promptAndPay = payService.payWithConfirm(sessionPay);
        prompt = (Prompt) promptAndPay[0];
        Pay pay = (Pay) promptAndPay[1];
```

```
        if (prompt != Prompt.success) {
            resp.data.prompt = prompt;
            return resp;
        }

        PayResp payResp = new PayResp();
        payResp.prompt = prompt;
        payResp.payOverview = PayOverview.fromPay(pay);

        //remove data in redis
        sessionPayService.delete(sessionPay.sessionId);

        //assign the payment result to paySyn data
        paySemaphore.paySynData.payResp = payResp;

        //release the semaphore in order to inform phoneScan
        paySemaphore.isFinished.release();

        resp.data.prompt = prompt;
        resp.data.payId = pay.payId;
        return resp;
    } catch (Exception e) {
        e.printStackTrace();
        resp.data.prompt = Prompt.unknownError;
        return resp;
    }
}
```

The method following is in service layer and mainly tries to persistent a payment involved in session payment. If it finished successfully, the returned result will contain the newly generated payment record – otherwise, a error message is returned.

```
public Object[] payWithConfirm(SessionPay sessionPay) {

    Prompt prompt = Prompt.pay_error;
    Prompt[] returnedPrompt = new Prompt[]{prompt};
    AtomicReference<Pay> pay = new AtomicReference<>();

    try {
        transactionHandler.runInTransactionSerially(() -> {
            pay.set(_pay(sessionPay.userId, sessionPay.merchantId, sessionPay.amount,
sessionPay.paymentPassword,
                    sessionPay.remarks, returnedPrompt));
```

```
            return null;
        });
    } catch (Exception e) {
        e.printStackTrace();
    }

    prompt = returnedPrompt[0];

    return new Object[]{prompt, pay.get()};
}
```

The following codes are relevant to transfer. The method makes a transfer according source user id, target user id, amount, payment password and remarks. After execution, the method will return the transfer state and transfer overview.

```
@ApiImplicitParams({
        @ApiImplicitParam(name = "token", paramType = "header"),
})
@PostMapping("/api/transfer")
public ResponseData<TransferResp> transfer(@ApiIgnore @RequestAttribute("userId") int
sourceId,
                                           @RequestParam(name = "targetUserId") int targetUserId,
                                           @RequestParam(name = "amount") BigDecimal amount,
                                           @RequestParam(name = "paymentPassword") String
paymentPassword,
                                           @RequestParam(name = "remarks") String remarks) {

    amount = amount.setScale(4, RoundingMode.HALF_UP);
    ResponseData<TransferResp> responseData = new ResponseData<>();
    responseData.data = new TransferResp();

    try{
        var promptAndTransfer = transferService.transfer(sourceId, targetUserId, amount,
paymentPassword, remarks);
        responseData.data.prompt = (Prompt) promptAndTransfer[0];
        responseData.data.transfer = (Transfer) promptAndTransfer[1];
    } catch (Exception e) {
        e.printStackTrace();
        responseData.errorPrompt = "Error";
        responseData.status = ResponseData.ERROR;
    }
    return responseData;
}
```

The following code is the transfer in service layer. It calls _transfer method and return the execution state and generated transfer record.

```
public Object[] transfer(int sourceUserId, int targetUserId, BigDecimal amount,String
paymentPassword,String remarks) {

   Prompt prompt = Prompt.transfer_error;
   Prompt[] returnedPrompt = new Prompt[]{prompt};
   Transfer transfer = null;


   try{
      transfer = transactionHandler.runInTransactionSerially(
            () ->
_transfer(sourceUserId,targetUserId,amount,paymentPassword,remarks,returnedPrompt));
   }catch (Exception e) {
      e.printStackTrace();    }
   prompt = returnedPrompt[0];
   return new Object[]{prompt,transfer};}
```

The following code segment tries to insert a transfer with given parameters. If successfully finishes, the generated transfer record will be returned – otherwise, null will be returned.

```
private Transfer _transfer(int sourceUserId,int targetUserId,BigDecimal amount,String
paymentPassword,
                  String remarks,
                  Prompt[] returnedPrompt) {


   User sourceUser = userDao.selectById(sourceUserId);
   User targetUser = userDao.selectById(targetUserId);

   LambdaLogicChain<Prompt> logicChain = new LambdaLogicChain<>();

   returnedPrompt[0] = logicChain.process(
     () -> sourceUser == null? Prompt.transfer_source_not_exist:null,

     () -> targetUser == null? Prompt.transfer_target_not_exist:null,

     () -
> !sourceUser.paymentPassword.equals(paymentPassword)?Prompt.payment_password_not_c
orrect:null,
```

```
        () -> sourceUserId == targetUserId?Prompt.transfer_to_self_error:null,
        () -> amount.compareTo(BigDecimal.ZERO) <=
0?Prompt.transfer_amount_invalid_error:null,
        () -> sourceUser.state == State.frozen?Prompt.transfer_source_account_frozen:null,
        () -> sourceUser.state ==
State.unverified?Prompt.transfer_source_account_unverified:null,
        () -> targetUser.state == State.frozen?Prompt.transfer_target_account_frozen:null,
        () -> targetUser.state ==
State.unverified?Prompt.transfer_target_account_unverified:null,
        () -> sourceUser.moneyAmount.compareTo(amount) < 0?
Prompt.transfer_not_enough_balance:null,
        () -> Prompt.transfer_error
    );

    if (returnedPrompt[0] != Prompt.transfer_error) {
        throw new RuntimeException(returnedPrompt[0].prompt);
    }
```

The following code removes the user balance and update the database

```
    sourceUser.moneyAmount = sourceUser.moneyAmount.subtract(amount);
    userDao.updateById(sourceUser);
```

The following code updates the target user's balance and generate a transfer.

```
    targetUser.moneyAmount = targetUser.moneyAmount.add(amount);
    userDao.updateById(targetUser);

    //generate a new transfer
    Transfer transfer = new Transfer();
    transfer.amount = amount;
    transfer.remarks = remarks;
    transfer.sourceUserId = sourceUserId;
    transfer.targetUserId = targetUserId;

    transferDao.insert(transfer);

    returnedPrompt[0] = Prompt.success;
```

```
    return transferDao.selectById(transfer.transferId);
}
```

The following codes are related with payment refund.This method needs *userId* and *paymentId* as parameters, check whether the merchant with given *userId* has right to access the payment, executes the refund of the payment and eventually response the state indicating the refund process.

```
@PutMapping("/api/payment/state")
public ResponseData<Prompt> refundPay(@RequestAttribute("userId") int userId,
@RequestBody int paymentId) {

    ResponseData<Prompt> responseData = new ResponseData<>();

    try {
        Merchant merchant = merchantService.getMerchantByUserId(userId);
        responseData.data = payService.refundPayWithMerchantId(merchant.merchantId,
paymentId);
        return responseData;
    } catch (Exception e) {
        e.printStackTrace();
        responseData.data = Prompt.unknownError;
        return responseData;
    }
}
```

The method following is similar to refund above and merchant Id, payment Id is encrypted in parameter. It returns the state indicating whether the refund succeeds.

```
@PostMapping("/api/refund")
public ResponseData<Prompt> refundPayUsingRSA(@RequestBody String
RSAEncryptedBase64String) {
    ResponseData<Prompt> responseData = new ResponseData<>();
    try {
        RefundRequest refundRequest = refundService.extractInfo(RSAEncryptedBase64String);

        if (refundRequest == null) {
            responseData.data = Prompt.refund_wrong_request_format;
            return responseData;
        }

        Prompt prompt = refundService.validateSignature(refundRequest);
```

```
        if (prompt != Prompt.success) {
            responseData.data = prompt;
            return responseData;
        }

        Merchant merchant = merchantService.getMerchantById(refundRequest.merchantId);

        return refundPay(merchant.merchantUserId, refundRequest.payId);
    } catch (Exception e) {
        e.printStackTrace();
        responseData.data = Prompt.unknownError;
        return responseData;
    }
}
```

This method check whether the given payment is paid to the given merchant. If so, it invokes refund transaction and return the execution state – otherwise, error 'not enough right' is returned.

```
public Prompt refundPayWithMerchantId(int merchantId, int payId) {

    LambdaLogicChain<Prompt> chain = new LambdaLogicChain<>();

    Merchant m = merchantDao.selectById(merchantId);
    Pay p = payDao.selectById(payId);


    Prompt prompt = chain.process(
            () -> m == null ? Prompt.refund_pay_merchant_not_exist : null,
            () -> p == null ? Prompt.refund_pay_id_not_exist : null,
            () -> p.targetMerchantId != merchantId ?
Prompt.refuned_pay_merchant_not_enough_right_refund : null);
    if (prompt != null)
        return prompt;

    prompt = refundPay(payId);
    return prompt;
}
```

This method refunds the given payment. It first check several conditions, then start a transaction to do refund data operation and finally return the transaction state.

```
public Prompt refundPay(int payId) {
    try {
        return transactionHandler.runInNewTransactionSerially(() -> {
            Pay pay = payDao.selectById(payId);
            if (pay == null)
                return Prompt.refund_pay_id_not_exist;
            else if (pay.state == PayState.refunded)
                return Prompt.refund_pay_id_already_refunded;
            User user = userDao.selectById(pay.sourceUserId);
            Merchant merchant = merchantDao.selectById(pay.targetMerchantId);
```

The following code compute the amount supposed to be refunded and remove the balance from the merchant account.

```
            BigDecimal amount = pay.amount.subtract(pay.fee);
            merchant.moneyAmount = merchant.moneyAmount.subtract(amount);
            merchantDao.updateById(merchant);
```

The following code add the user account the refunded balance, insert a refund record and change the status of the payment record

```
            user.moneyAmount = user.moneyAmount.add(pay.amount);
            userDao.updateById(user);

            //insert a refund record
            Refund refund = new Refund();
            refund.payId = payId;
            refundDao.insert(refund);

            //change the state of the pay
            pay.state = PayState.refunded;
            payDao.updateById(pay);

            return Prompt.success;
        });
    } catch (Exception e) {
        e.printStackTrace();
        return Prompt.unknownError;
    }
}
```

4 SYSTEM DEVELOPMENT AND MANUAL

4.1 User's Guide

The following section mainly illustrates the user's guide on android app.

When the user first opens the app after installation, the login page shows (figure 4.1). To login in the system, the user must input the correct phone number and password.

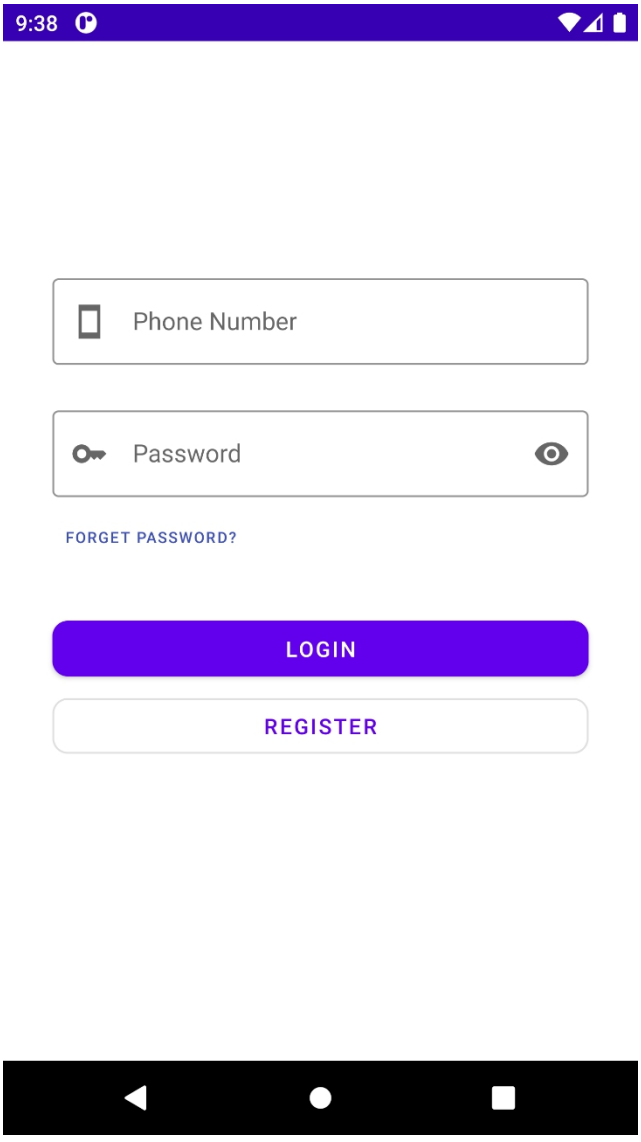


Figure 4.1 – The Login Page

For the first time, obviously, the user doesn't have an account and he/she may click the REGISTER button. Once clicking, the register page will show (figure 4.2). The first step of registration requires the user to input his first name, last name, nationality, ID number and ID document photo.

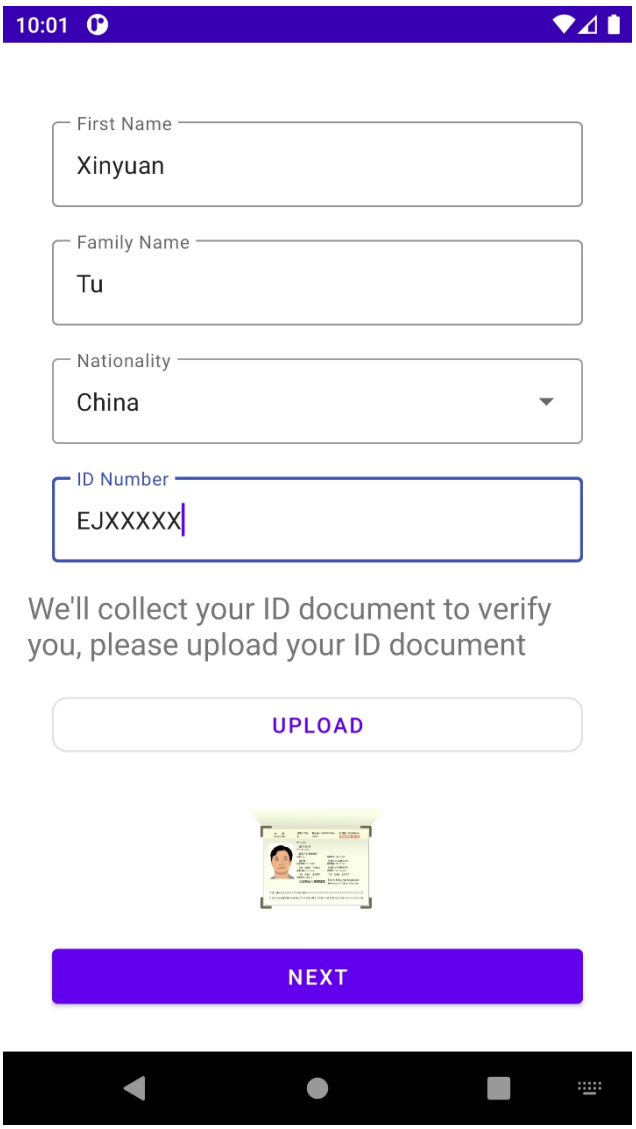


Figure 4.2 – First Step of Registration

Once finish all the text fields, click next button. Second step will show. In the second step, the user is required to verify his phone number and email (figure 4.3 left) .

When the user input the phone code and number, he/she can click send button, which later his/her mobile phone will receive the verify code. Similarly, for the verification of email, the verify code will be sent. After Filling all the fields correctly, the user can click next.

Then it will come to third step (figure 4.3 right) of registration afterwards, where the user is required to set the password for login and payment. Note that each password will typed twice to make sure good memorize.

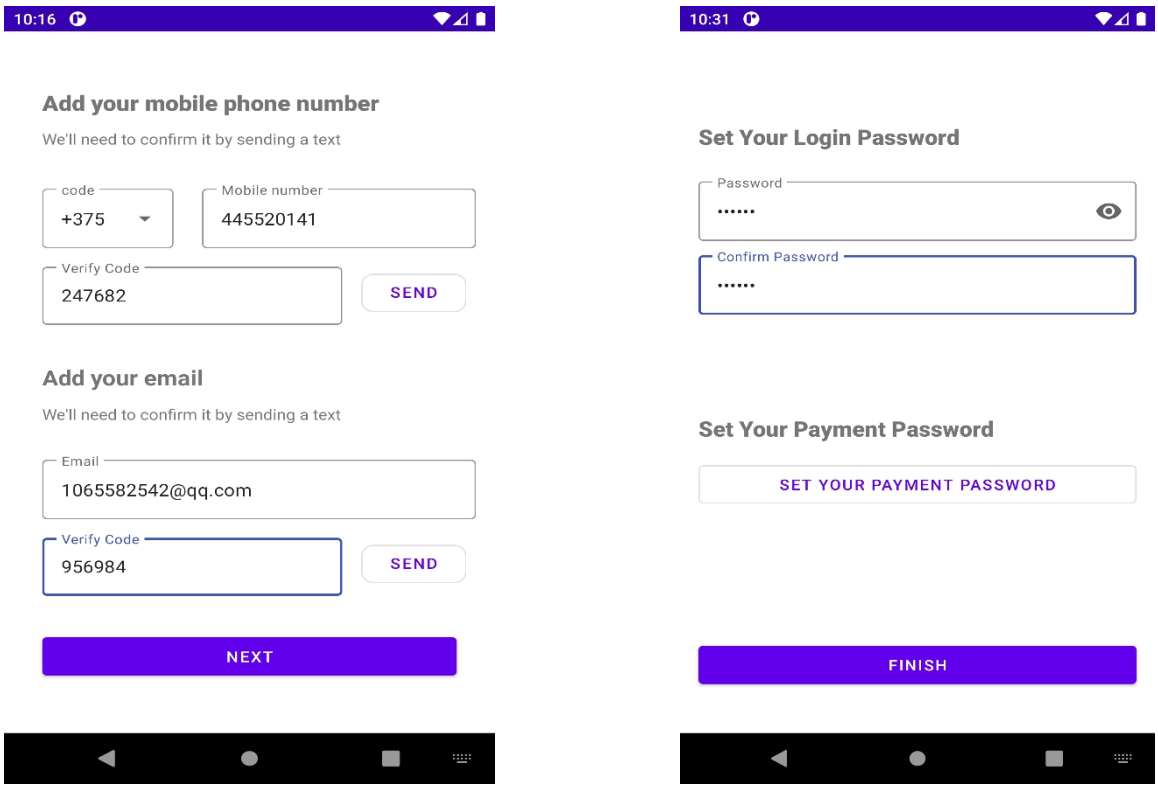


Figure 4.3 – Second Step (Left), Third Step (Right) of Registration

When both passwords are same, the user can click finish button. The final page indicating finish will show(figure 4.4).

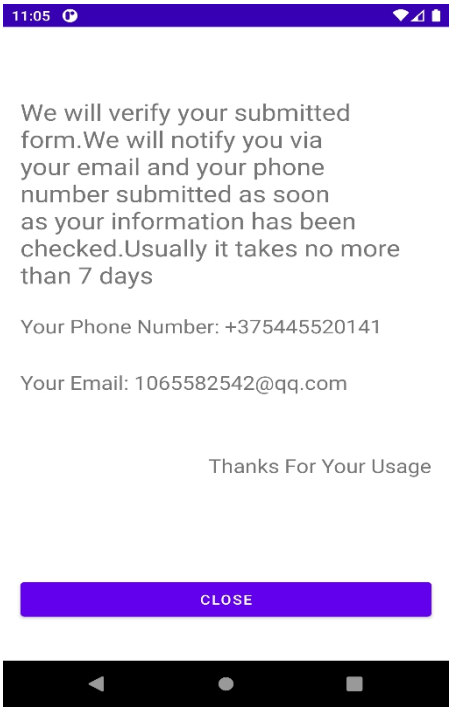


Figure 4.4 – Final Step of Registration

After the administrator verifies the registration, the user can login into the system and the home page will show (figure 4.5).

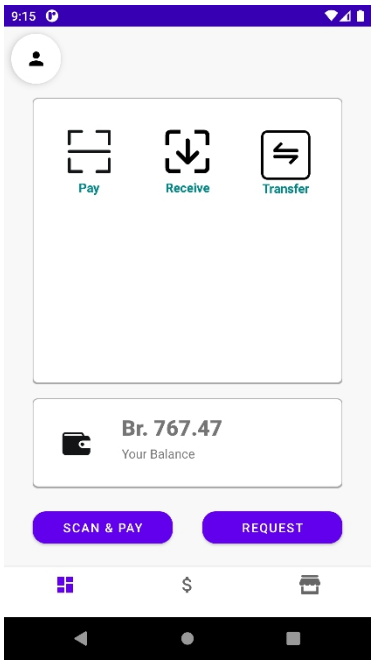


Figure 4.5 – Home Page

In the home page, the user can pay, receive and transfer funds. Also, the user can also see the detailed individual wallet (figure 4.6 left) by pressing the second bottom navigation button. Furthermore, the third navigation button switches to detailed merchant wallet (figure 4.6 right).

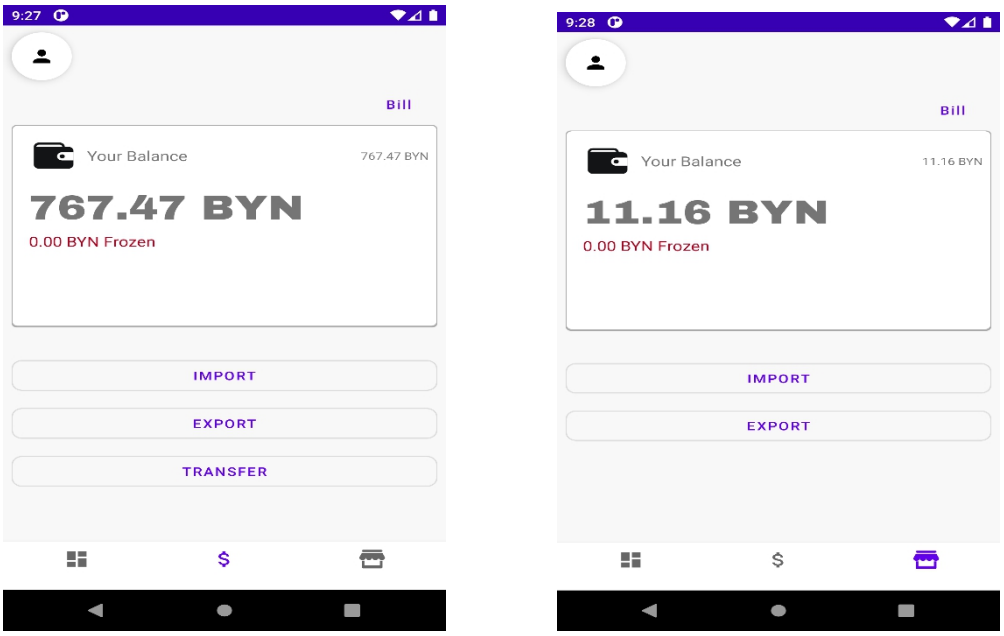


Figure 4.6 – Individual Wallet (Left), Merchant Wallet (Right)

In the individual wallet page, the user can see bills, import their funds, export their funds and transfer. In the merchant wallet page, bills, import funds and export funds can be chosen

In the home page, if either Pay or SCAN & Pay is clicked, the app show open the camera(figure 4.7).

Figure 4.7– Camera Scanning Page

If the camera scans the someone's transfer or payment qr code, transfer page (figure 4.8 left) or payment page (figure 4.8 middle) will show.

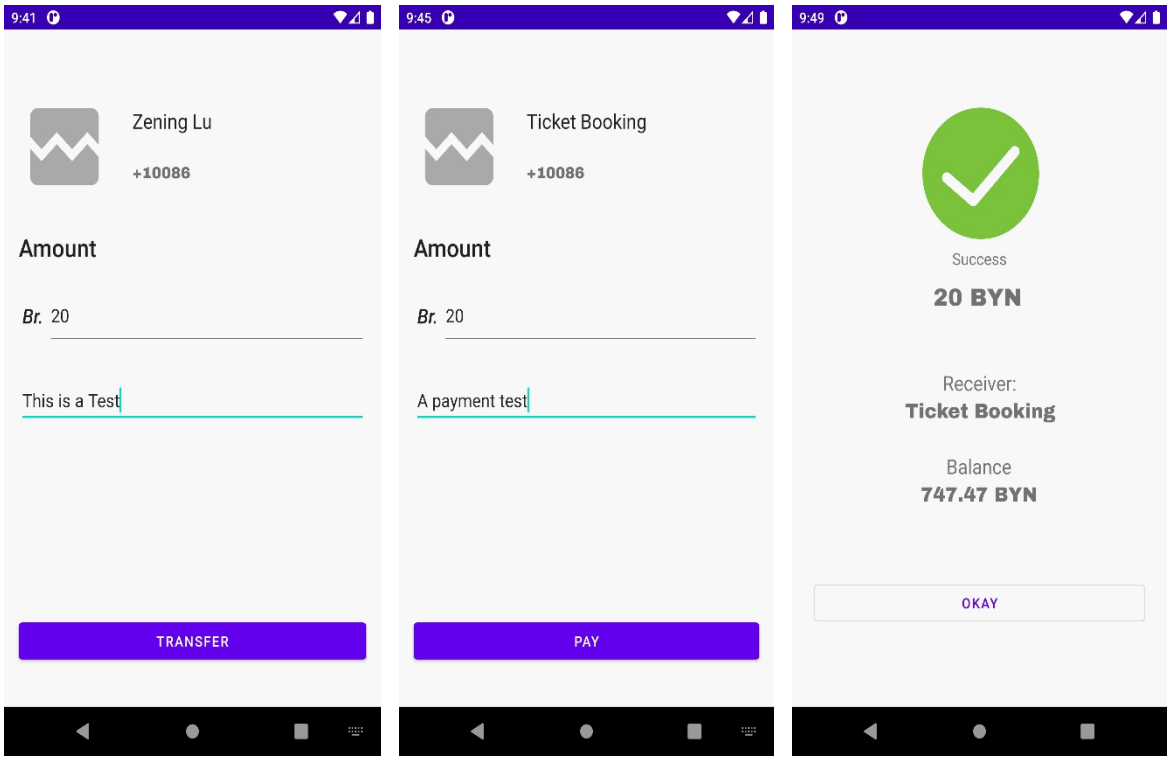


Figure 4.8 – Transfer Page (Left), Pay Page (Middle), Checkout Page (Right)

In the transfer page, amount must be completed and remarks can be filled. Clicking TRANSFER button, the app requires user to input payment password. If the password is correct and balance is enough, the transfer process will proceed. The payment process is the same with transfer.

After payment or transfer, the checkout page will show (see figure 4.8 right). The checkout page shows the status, amount, receiver of the transaction and the balances left.

If the user has a merchant account, he can click the switch button to receive funds to merchant wallet.In the home page, when the user click Receive, receive qr code will show(figure 4.9).

If the user has a verified merchant account, the SWITCH button is clickable – otherwise the button is not enabled until the user register a merchant account and the merchant is verified.

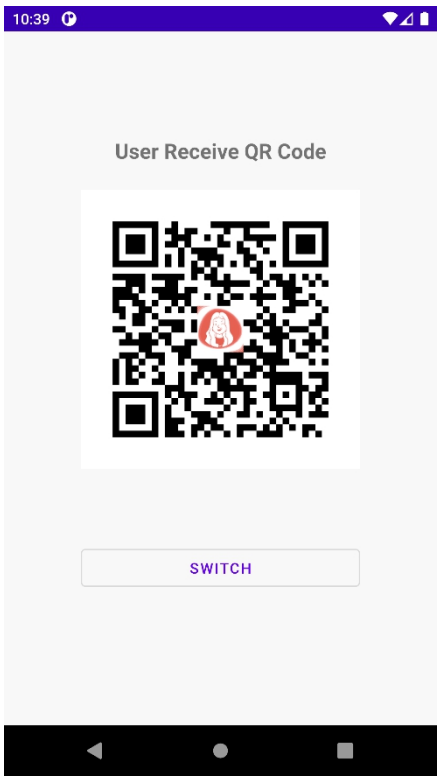


Figure 4.9 – Receive Page

In the individual wallet page, if the user clicks bill button on the right top, he/she can view bill list (figure 4.10)

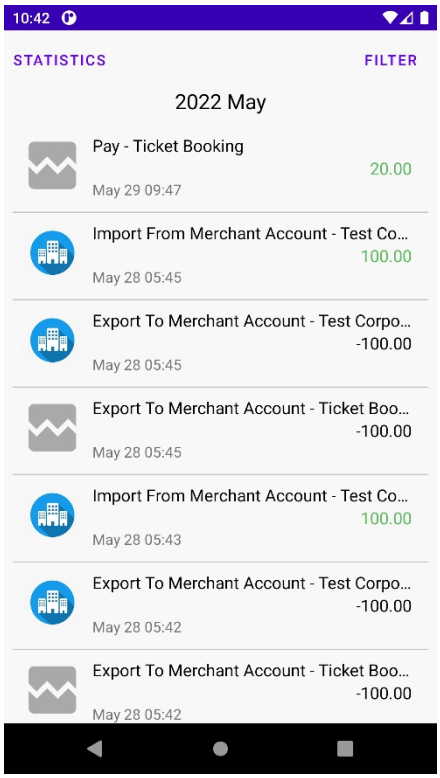


Figure 4.10 – Bill Page

In the bill page, the user scroll down to load more bills and it is also possible to set criteria by clicking the top right FILTER.

The system also have considered import and export, there are mainly four types of import and export patterns:

- From merchant account to individual account. Click Import button in individual wallet page (figure 4.6 left) and select Import from Merchant Account in Import select list, or click Export button in merchant wallet page (figure 4.6 right) and select Export to Individual Account in select list. Afterwards, the UI will show (figure 4.11 left).
- From individual account to merchant account. Click Export button in individual wallet page (figure 4.6 left) and select Export to Merchant Account in Export select list, or click Import button in merchant wallet page (figure 4.6 right) and select Import From Individual Account in select list. Afterwards, the UI will show (figure 4.11 right).
- Import funds from bank card. Click Import Button in either individual wallet page (figure 4.6 left) or merchant wallet page (figure 4.6 right). Next select import from credit card. The UI will switch to the credit card import page (figure 4.12)
- Export funds to bank card. Click Export Button in either individual wallet page (figure 4.6 left) or merchant wallet page (figure 4.6 right). Next select Export to Credit Card. The UI will switch to the credit card import page (figure 4.13).

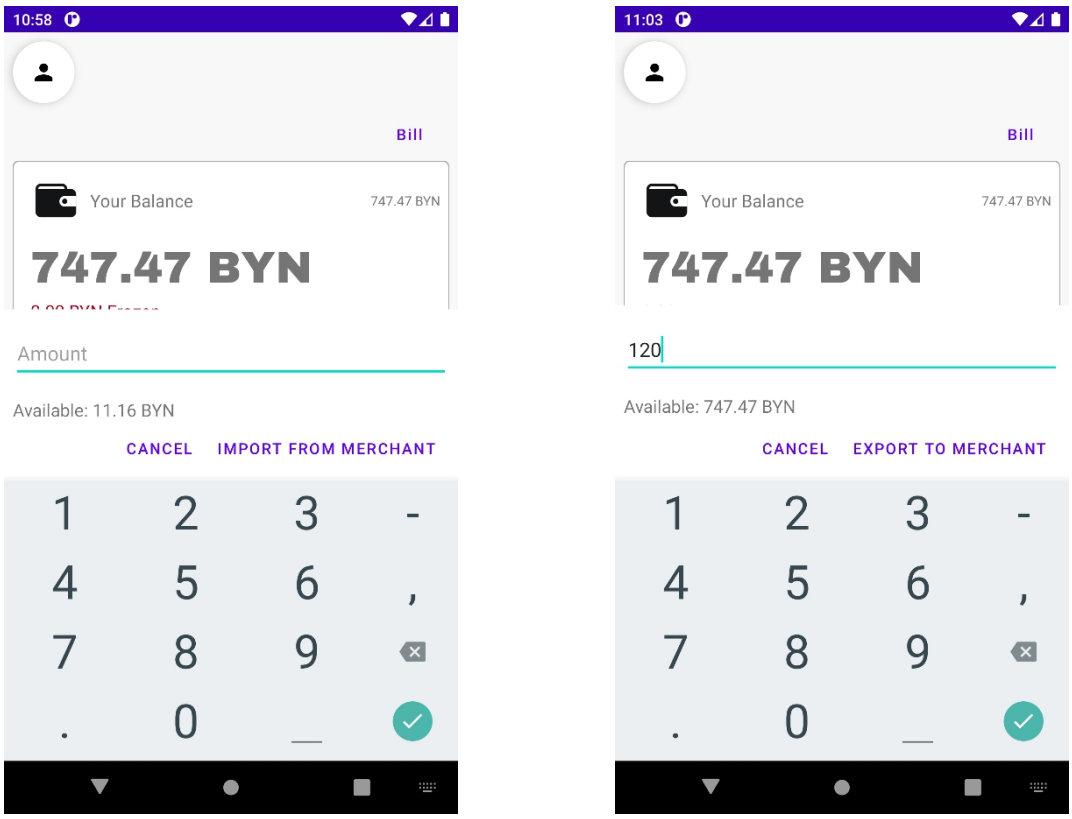


Figure 4.11 – Import from Merchant (Left), Export to (Right) Merchant

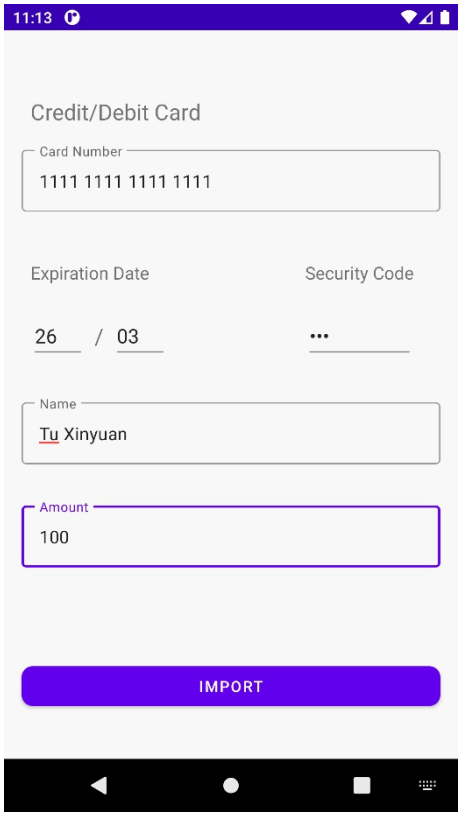


Figure 4.12 – Import Funds from Credit Card

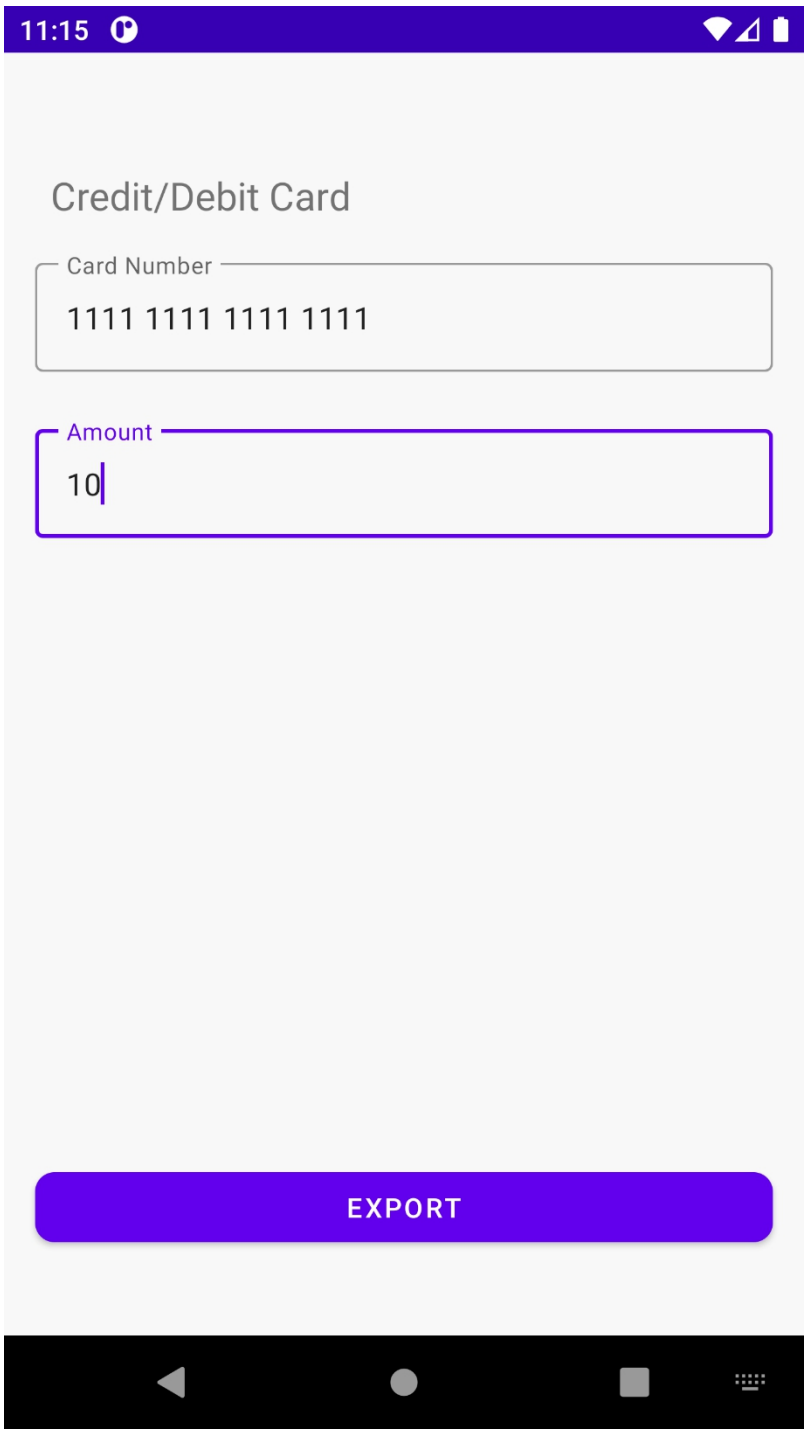


Figure 4.13 – Export Funds to Credit Card

In this page, the use is required to input the credit card number and amount. When filling completes, click the Export Button. After that, payment password dialog will pop up. If payment password is correct, the export procedure will proceed.

If the user doesn't have a merchant account, he/she can apply in merchant wallet page (figure 4.6 right). There should be a apply button in the center of the page(figure 4.14). Click the apply button, the first step of merchant registration page will show(figure 4.15).

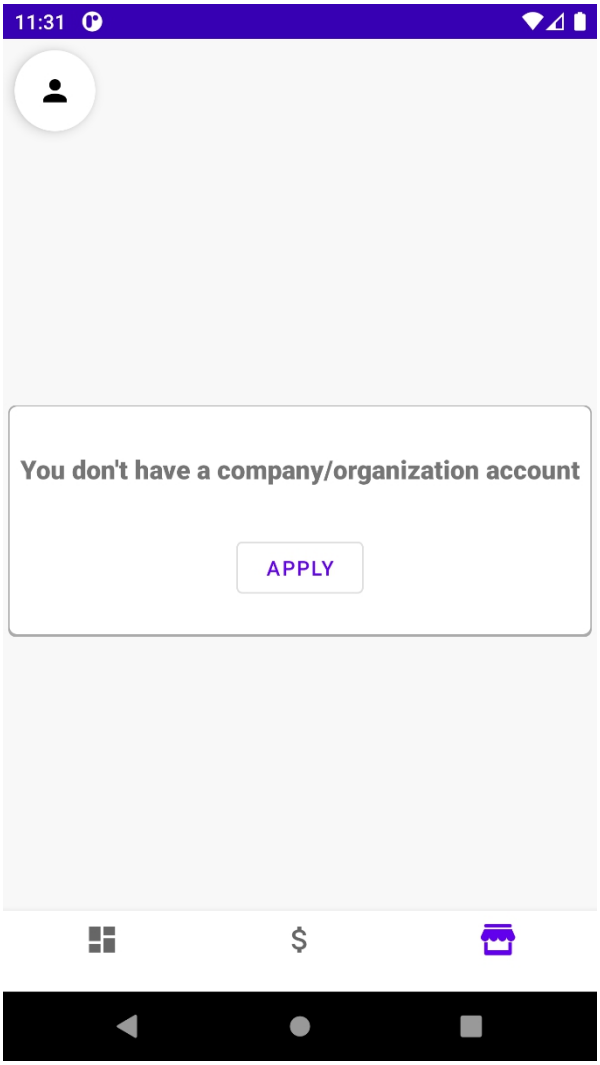


Figure 4.14 – Apply Page for User with no Merchant Account (Left)

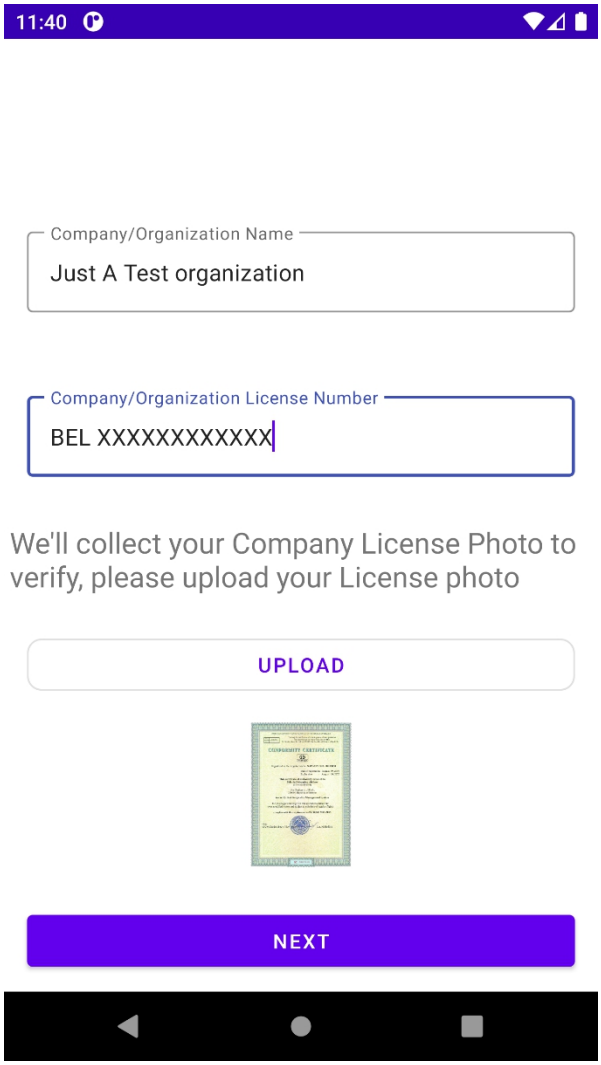
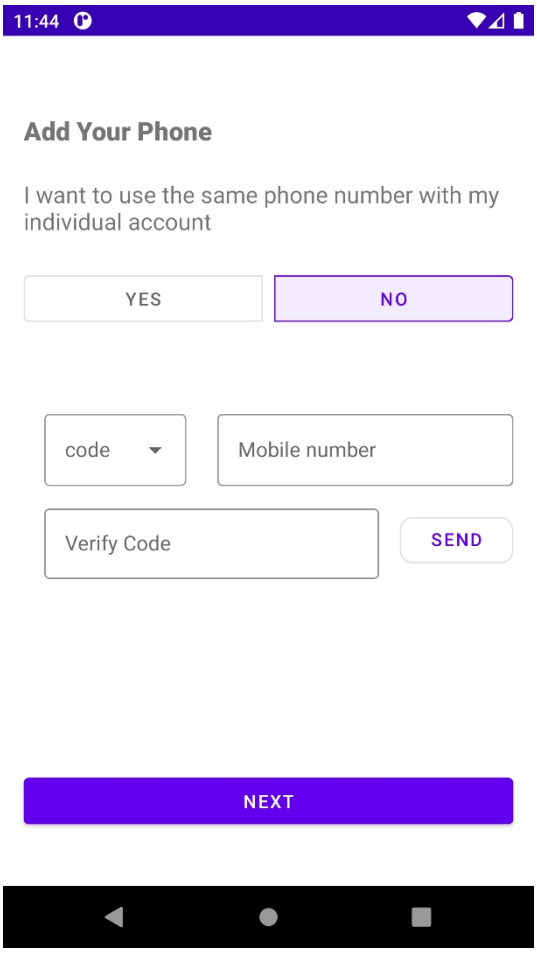


Figure 4.15 – First Step of Merchant Registration (Right)

In the first step of registration, the user is required to input the company or organization name, company or organization license number and upload the license photo.

When everything is finished, click the next button, and the second step page will show (figure 4.16).



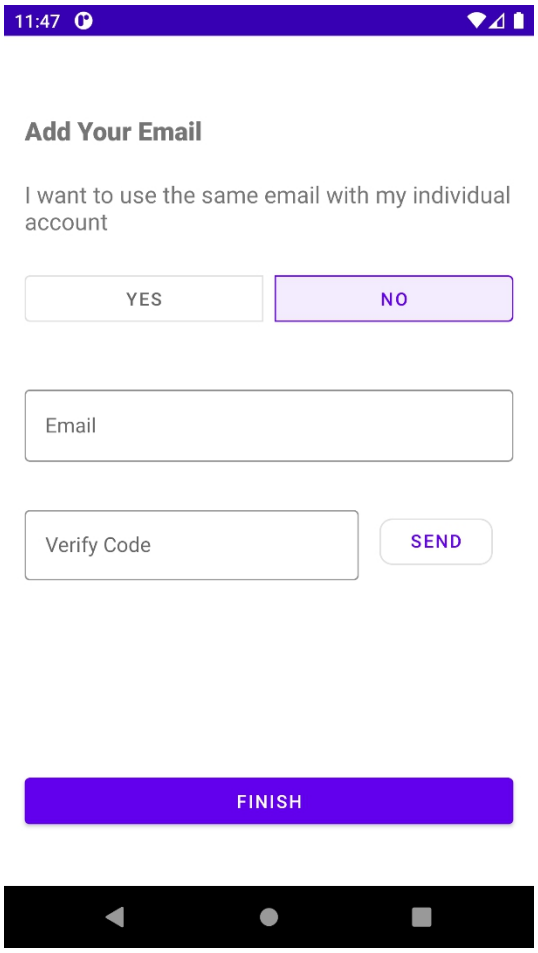


Figure 4.16 – Second Step of Merchant Registration (Left)

Figure 4.17 – Third Step of Merchant Registration (Right)

In the second step, the user is required to verify his phone number. He/she can choose to use the same phone number with individual account or use a different one. If choosing a different one, SMS verification is needed. Note that clicking SEND button each time disables it for one minute, and the SMS code will expire after three minutes. When everything is finished, click the next button, it will come to third step.

The third step is almost same to second step except it is for email verification. When filling the these forms, click the finish button will go to the final step(figure 4.18).

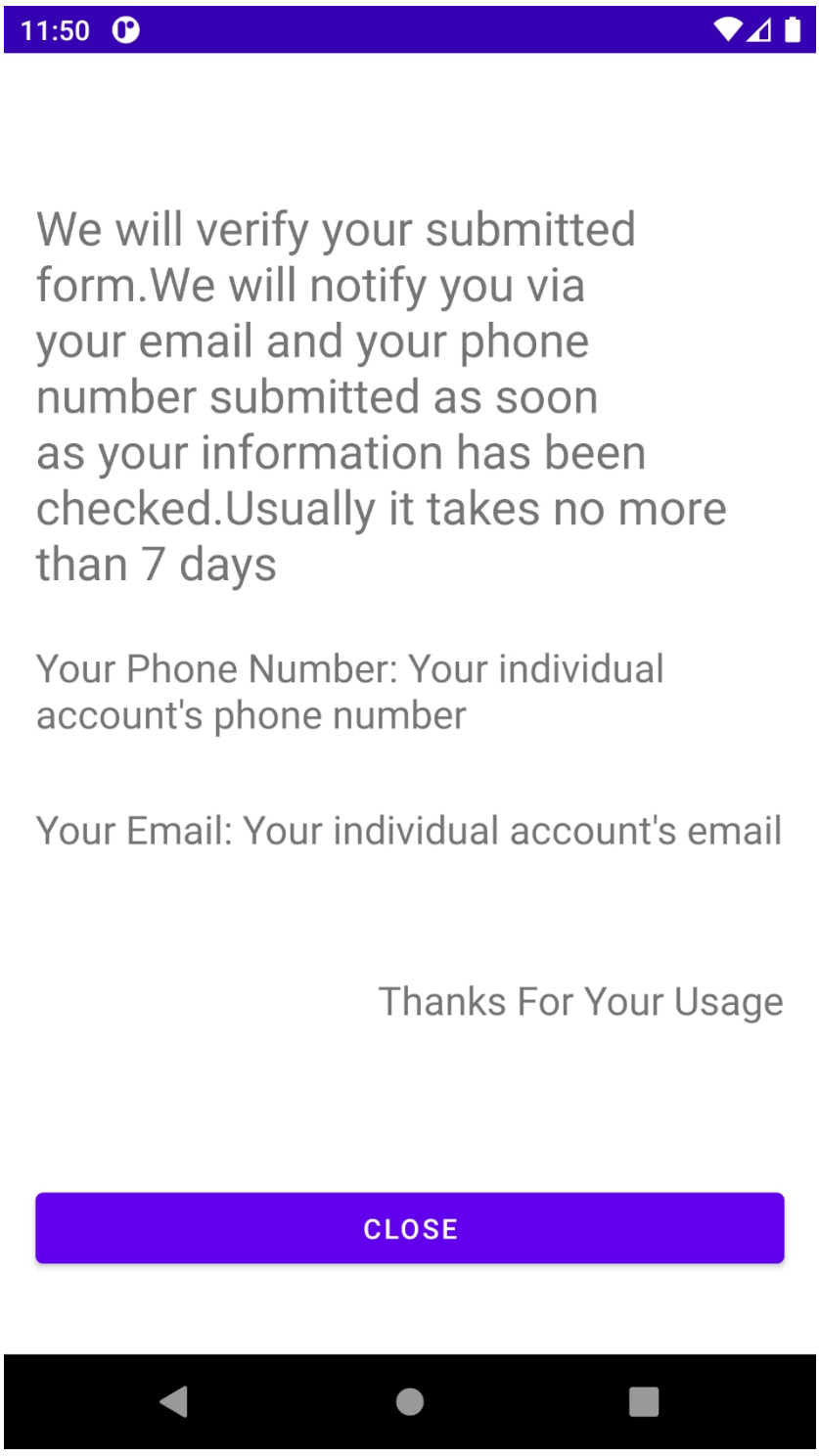


Figure 4.18 – Final Step of Merchant Registration

4.2 Administrator's Guide

The following section introduces the interactions between the system and administrators.

To login in the system, first open the browser and type the URL – http://34.118.47.158:8080/#/. The administrator's login page will show (figure 4.19).

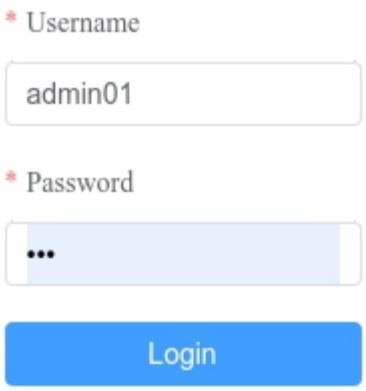


Figure 4.19 – Administrator's Login Page

After clicking the Login button, if successfully, the home page will show (figure 4.20).

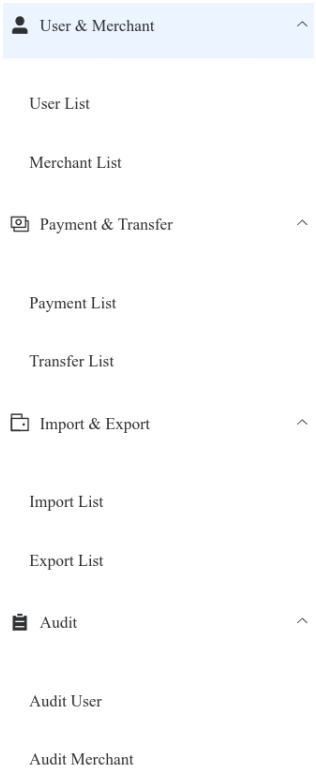


Figure 4.20 – Administrator's Home Page

In this page, click user list. The administrator can see the user list(figure 4.21), and freeze the user's balance or account if desired. By clicking the operation button on the user list, the operations such as freeze and unfreeze will show(figure 4.22).

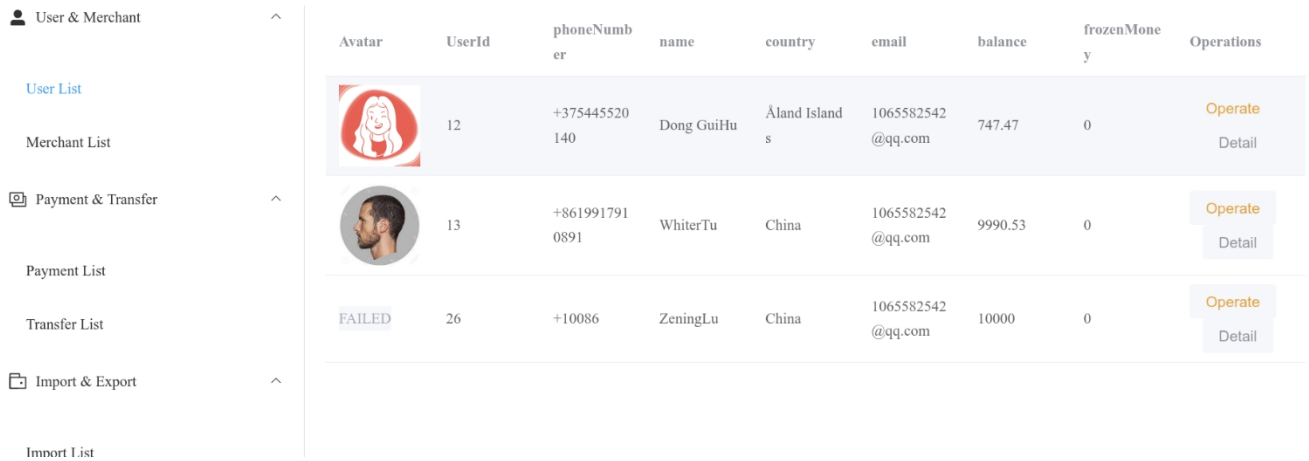


Figure 4.21 – User List Page

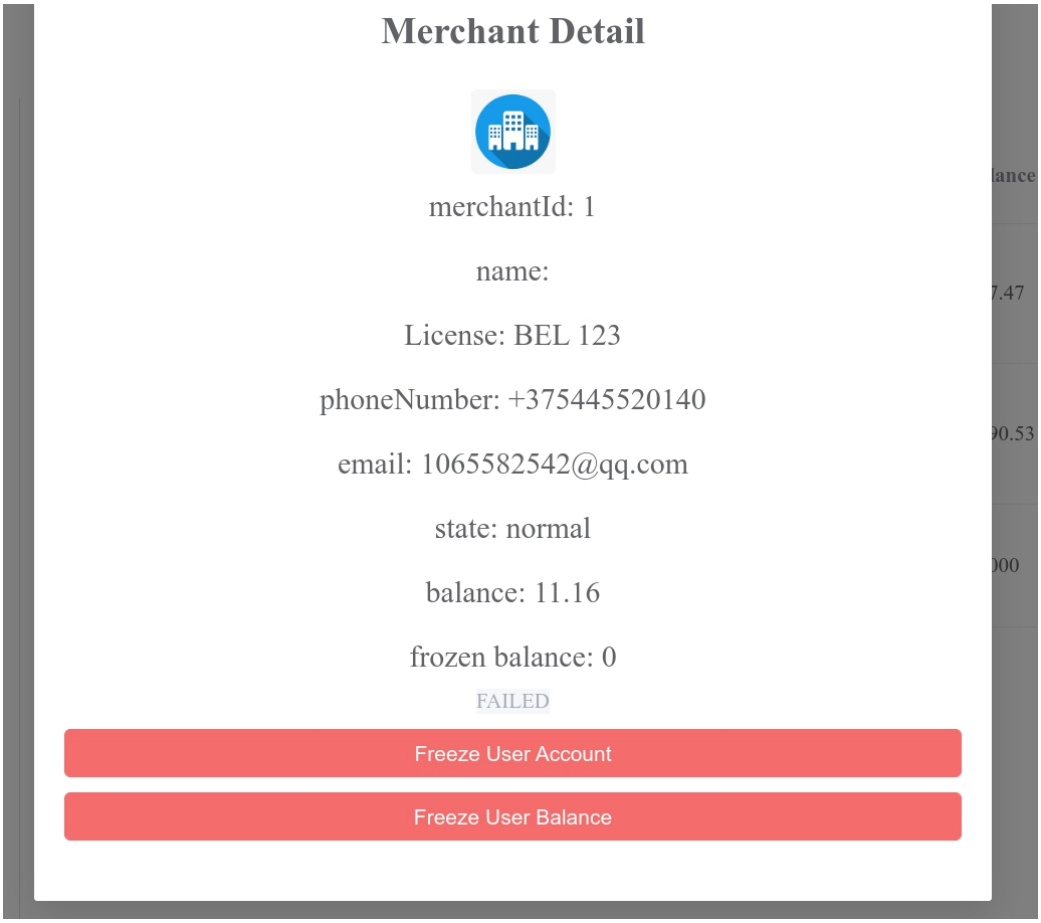


Figure 4.22 – Operation Page

The page for merchants is almost the same. The merchant list page is shown (figure 4.23), and the merchant operation page is shown (figure 4.24).

Figure 4.23 – Merchant List Page

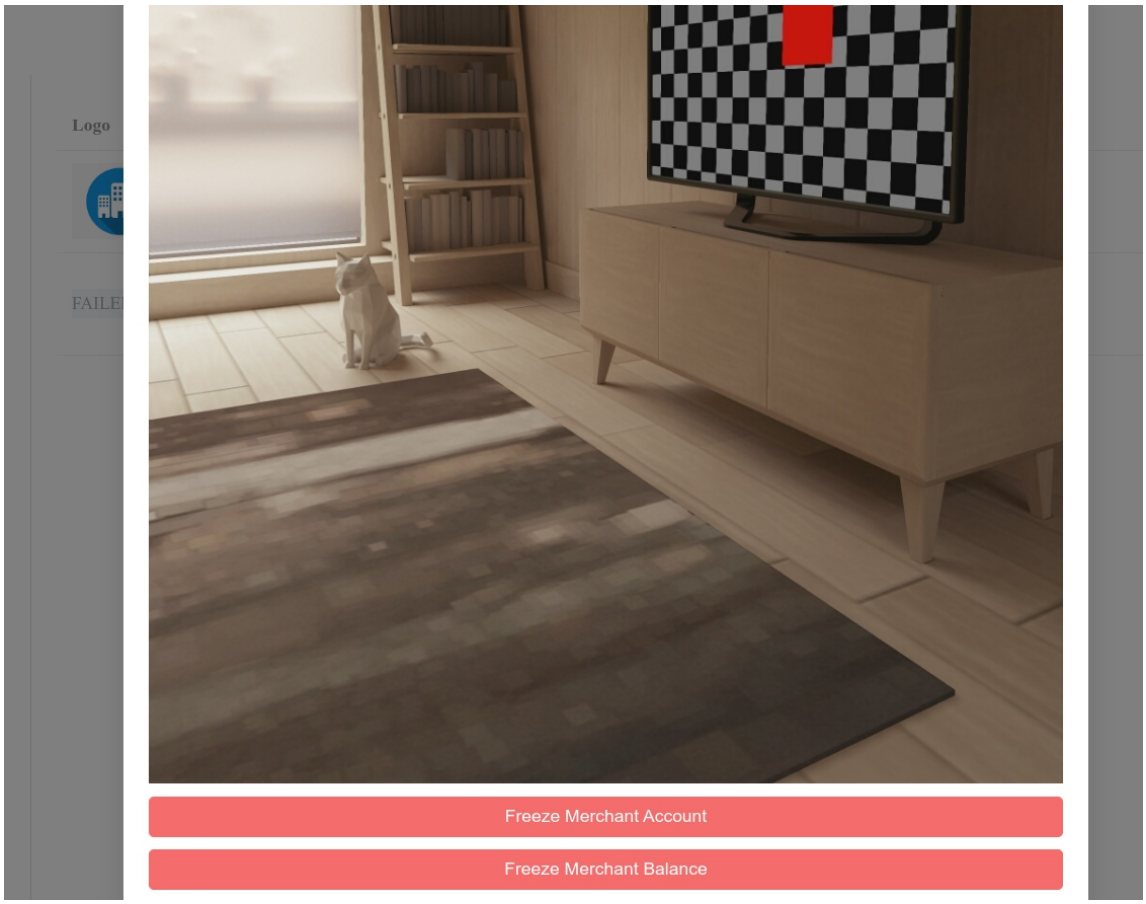


Figure 4.24 – Merchant Operation Page

When clicking the Payment List on the left menu, the administrator can view all the payment list (figure 4.25), and, clicking the Detail button for each record, the detail information can be seen (figure 4.26).

Figure 4.25 – Payment List Page

Figure 4.26 – Payment Detail Page

For other records, transfer, import and export, the procedure and UI are similar.

For auditing user, in other word, verification of user's registration, the page (figure 4.27) will show by clicking Audit User submenu on the left menu. Clicking the Operate button on table's record, the auditing user page (figure 4.28) will show.

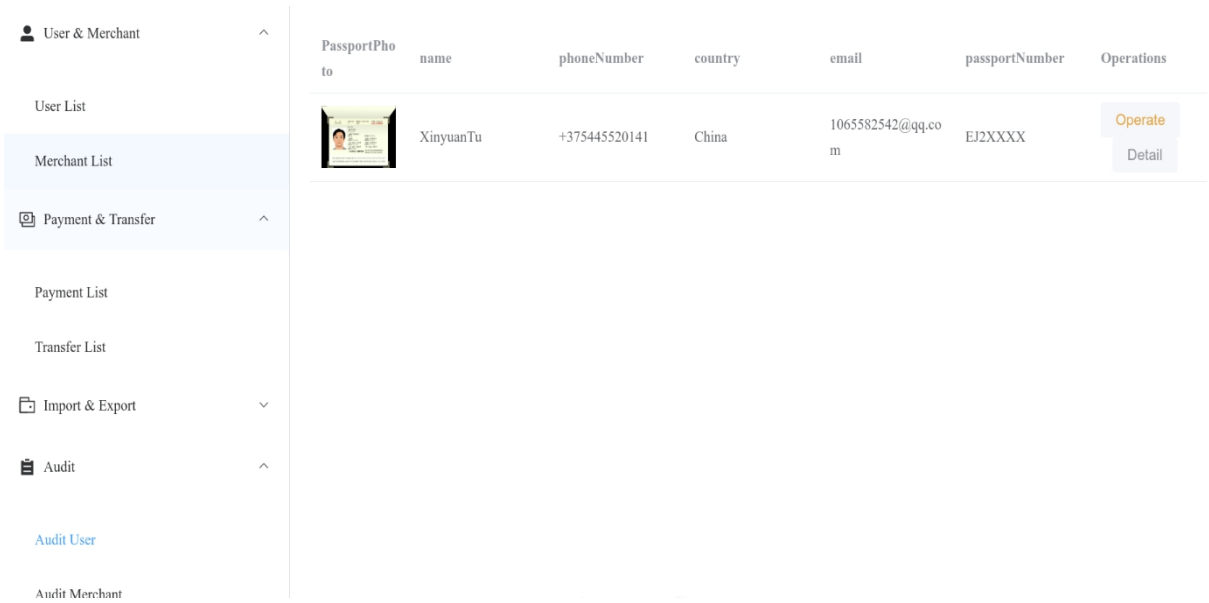


Figure 4.27 – Audit User List Page

Figure 4.28 – Audit User Page

For merchant audit, it is similar with the user's one. The following two figures show the audit merchant list page (figure 4.29) and audit merchant page (figure 4.30).

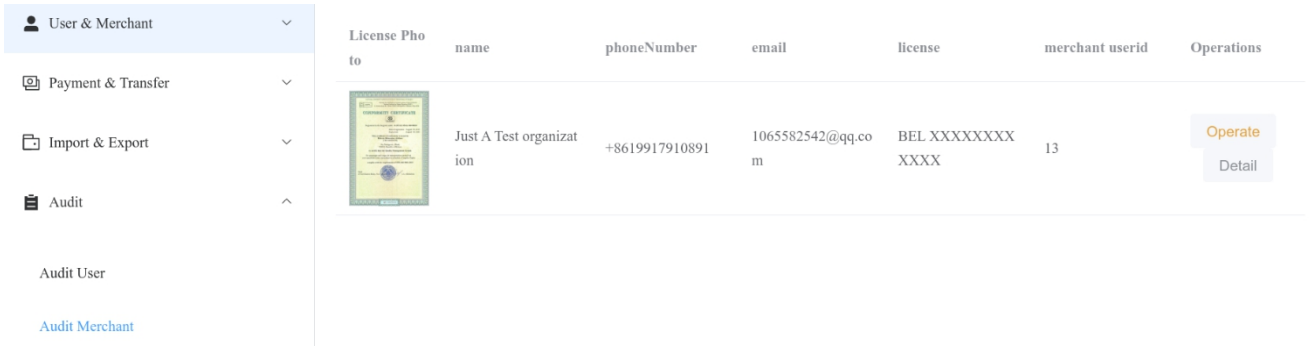


Figure 4.29 – Audit Merchant List Page

Figure 4.30 – Audit Merchant Page

4.3 Further Development

4.3.1 More Distribution Support

As per latest Google Play stats, you will be shocked to know that there are 3.48 million apps currently at the Google Play Store. This number of apps on Google Play is on a rise as 3,739 apps are added to the Play Store every single day[20]. So it's critical to upload the android app to Google Play if the developer wants the app to be more widely–used.

4.3.2 More Real Time Messaging Support

Though the system is not programmed to send asynchronous messaging, when transaction is made, this feature will enabled in the future, either by using Google Firebase messaging or developing the messaging server. Once equipped with real time messaging, the app would become more market–dominant.

4.3.3 More Bank Card Support

Currently, the app doesn't support interactions with outer banks. In the future, if the developer wants the app to be more flexible, freely exporting and importing should be implemented. Also, if it were implemented, the product would be dominant in the market since it would be more powerful and user friendly than the typical mobile payment app PayPal.

4.3.4 Distributed Framework Support

If the app becomes popular in the future, the network flow into and out the system will be large. To deal large network flow, regional distributed system will be considered. Also, to have high tolerance, especially in terms of server crush, computer cluster will be applied.

4.3.5 More Cooperation with Government

In China, the dominant mobile payment service providers, like AliPay and Wechat Pay, cooperate deeply with the government , and providers can send requests to National Citizen Data Center for face verification and ID verification. To determine user's identity effectively and automatically (without manual audit) , I'm looking forward to cooperating with the government for such services.

4.3.6 More Efficient Payment and Authentication Method

In the future, the system may be developed to support more bioidentification method. For example, flesh face recognition can be provided. To implement this, more data center will be established, and AI models in the computing server will be trained. The main server will communicate with other recognition systems.

5 ECONOMIC FEASIBILITY STUDY

5.1 Characteristic of an economic case of the project

Diploma project ‘Automated Mobile Payment System’ is an application which provides payment services for both merchants and users. The system relies on today’s widely used mobile phone as payment client. Downloading the application, users can start transactions with registered merchants while merchants can request payment from users. Furthermore, importing and exporting balance from the current common credits card system (eg. Mastercard, Visa) into the mobile payment system is also available.

5.2 Calculation of cost of materials for project accomplishment

The estimate of costs for carrying out of scientifically research work settles payments under following clauses. Calculation is performed under the formula:

$$Pm = Ktp \sum_{i=1}^{n} Hpi * Ci$$

Where; KTP – the coefficient considering hauling expenses
(KTP ≈ from 1.0 to 1.10) for the project we accept KTP = 1;
Hpi – norm of the expenses a material kind on the project;
Ci – unit of selling price of material kind, ruble;
N – Quantity of applied kinds of materials.

Table 5.1 − Calculation of costs for materials

№	The name of materials	Unit of measure	The price, ruble.	Quantity	The sum, ruble.
1	Paper format A1	Sheet	0.7	50	35
2	Paper format A4	Sheet	0.15	200	30
3	Stationery	–	–	–	70
4	Materials for experiences and designing	–	–	–	120
The sum of expenses		–	–	–	255

The estimate of costs for carrying out of scientifically research work settles payments under following clauses:

5.3 Calculation of a base salary of the personnel occupied with accomplishment of works under the project.

The size of costs settles payments under the formula:

$$Pow = Knp \sum_{i=1}^{n} Tci * Ni * ti$$

Where; Tci – a wage rate for a day, categories of workers, ruble;
Ni – quantity of workers of a category;
ti – time of actual work of the worker of a category under the project, day;
Knp – coefficient of awards on bonus systems
(Knp ≈ from 1.10 to 1.40) for the project we accept Knp = 1.2;
Calculation of the produce in the table:

Table 5.2– Base salary calculation

№	The name of categories of workers and posts	Quantity of units, the people	Salary for one month, ruble.	Coefficient of bonus surcharges	Expenditures of labour, months	The sum, ruble.
1	The supervisor of studies of the project	1	1000	1.2	3	3600
2	The engineer	2	750	1.2	3	5400
3	UI designer	1	800	1.2	3	2880
The sum of expenses		–	–	–	–	11880

5.4 Calculation of an additional salary of the contractors

Calculation of an additional salary of the contractors, including the various payments provided by the labour law, under the formula:

Additional wages include a variety of performers stipulated by the labour legislation of the payment and is calculated according to the formula:

$$Pnw = Pow * \frac{Hnw}{100}$$

Where, Hnw – the specification of an additional salary,
Hnw ≈ from 10 to 25%, for the project it is accepted Hnw = 25%.

$$Pnw = 11880 * \frac{25}{100} = 2970 \text{ ruble}$$

5.5 Calculation of deductions to social insurance

Calculation of deductions to social insurance under the formula:

$$\text{Poc} = (\text{Pow} + \text{Pnw}) * \frac{\text{Hoc}}{100}$$

Where, Hoc – rate of deductions on social ensurance (tax), Hoc = 30.0%

$$\text{Poc} = (11880 + 2970) * \frac{30.0}{100} = 4455 \text{ ruble}$$

5.6 Calculation of expenses on scientific business trip

We calculate the other expenses for materials scientific and technical information and the fee for the use of internet and telephone, etc.

The cost is calculated according to the formula:

$$\text{P}kom = Pow * \frac{Hkom}{100}$$

Where, Hkom – the specification on scientific business trip expenses,

Hkom ≈ from 5 to 20%, for the project we accept Hkom = 20%.

$$\text{P}kom = 11880 * \frac{20}{100} = 2376 \text{ ruble}$$

5.7 Calculation of common enterprise expenses

Calculation of common enterprise expenses under the formula:

Indirect cost includes the cost of management and overhead cost, calculated according to the formula"

$$\text{P}koc = Pow * \frac{Hkoc}{100}$$

Where, Hkoc – the specification of indirect expenses, Hkoc ≈ from 50 to 100 %, for the project it is accepted Hkoc = 90 %

$$\text{P}koc = 11880 * \frac{90}{100} = 10692 \text{ ruble}$$

5.8 Calculation of the complete cost value of the project

The total cost of scientific and technical products is determined as the sum of all cost in all respects (clauses 1–6) as according to the formula:

$$C_n = P_m + P_{ow} + P_{nw} + P_{oc} + P_{kom} + P_{koc}$$

$$C_n = 255 + 11880 + 2970 + 4455 + 2376 + 10692 = 32628 \text{ ruble}$$

On level of profitability in percentage of the complete cost value the profit settles payments:

At the average level of profitability in percent of the total cost is determined by the target profit unit of scientific and technical products according to the formula:

$$Pr = Cn * \frac{Yp}{100}$$

Where, Yp – profitability level, Yp ≈ from 10 to 30 %, for the project we accept Yp = 20 %.

$$P_r = 32628 * \frac{20}{100} = 6525.6 \text{ ruble}$$

5.9 Calculation of the price of the project

Calculation of the price of the project under the formula:

To determine an approximate (estimated) wholesale price of scientific and technical products according to the formula,

$$Bn = Cn + Pr$$

$$B_n = 32628 + 6525.6 = 39153.6 \text{ ruble}$$

5.10 Calculation of the tax to value added (VAT)

Calculation of the tax to value added (VAT) under the formula:

The Value Added Tax is determined by the formula:

$$VAT = Bn * \frac{Hvat}{100}$$

Where, Hvat – the tax rate on vat (the tax), Hvat = 20%.

$$VAT = 39153.6 * \frac{20}{100} = 7830.72 \text{ ruble}$$

5.11 Calculation of the price of the project

Calculation of the price of the project taking into account the VAT under the formula: to determine the selling price of scientific and technical products with VAT according to the formula:

$$B = B_n + VAT$$

$$B = 391.53 + 7830.72 = 46984.32 \text{ ruble}$$

Calculation of costs for the project and the project price are resulted in table 5.3.

Table 5.3 − The Estimate of costs for the project

№	Clauses of costs	Calculation	The sum, ruble.
1	Materials (P_m)	Table 1	255
2	Base salary (P_{ow})	Table 2	11880
3	The additional salary (P_{nw})	$11800 * \frac{25}{100}$	2970
4	Deductions in population social insurance fund (P_{oc})	$(11800 + 2970) * \frac{30.0}{100}$	4455
5	Scientific business trip expenses (P_{kom})	$11800 * \frac{20}{100}$	2376
6	Common enterprise expenses (Ркос)	$11800 * \frac{90}{100}$	10692
7	Total the cost value (Cn)	255 + 11880 + 2970 + 4455 + 2376 + 10692	32628
8	Profit (P_r)	$32628 * \frac{20}{100}$	6525.6
9	The project price (B_n)	32628 + 6525.6	39153.6
10	The value−added tax (VAT)	$39153.6 * \frac{20}{100}$	7830.72
11	The price from the VAT (B)	39153.6 + 7830.72	46984.32

5.12 Economy Feasibility Study Conclusions

Automated Mobile Payment System where users can conduct secure mobile transactions and merchants can receive transactions has been achieved. Costs for development of such system have constituted 46984.32 ruble.

CONCLUSION

In the essay, the Automated Mobile Payment System has been analyzed, designed and implemented in a standard procedure. The Automated Mobile Payment System has combined the most advantages of existing systems and has reduced the shortcomings to the least. The system's target has been clarified.

In terms of design, functional diagrams and UML diagrams have been design solving most of the systems' business logic. Illustration of algorithm has been mentioned for private information protection. Databases are designed and implemented fitting appropriate normal forms. System borders and interactions are clear. The system designs are compatible with the demand description.

The system provides distinguished ergonomics both for users and administrators. It has high robustness when dealing with high network flow. Though deployed in single machine, the system is flexible and is designed to have high concurrency handling ability. It can be put into commercial use as it has fit the local law and regulations.

In general, the system is perfectly designed and implemented, which meets the users' satisfaction.

REFERENCES

[1] Laetitia, CHAIX; Dominique, TORRE. "Download Limit Exceeded". citeseerx.ist.psu.edu. University Nice Sophia–Antipolis. CiteSeerX 10.1.1.460.10. Retrieved 26 August 2021.

[2] Ahmady, Gholam & Mehrpour, Maryam & Nikooravesh, Aghdas. (2016). Organizational Structure. Procedia – Social and Behavioral Sciences. 230. 10.1016/j.sbspro.2016.09.057.

[3] Bartering Overview (PDF) [Electronic resource] – Access Mode: http://www.crbs.umd.edu/crossingborders/lessonplans/2006/cuseo – fields/cuseo – fileds_worksheets.pdf

[4] History of Money and Payments [Electronic resource] – Access Mode: https://squareup.com/us/en/townsquare/history–of–money–and–payments

[5] Risk Management Examination Manual of Credit Card Activities [Electronic Resource] – Access Mode: https://www.fdic.gov/regulations/examinations/credit_card/pdf_version/ch2.pdf

[6] Presley, Adrien & Liles, Donald. (1998). The Use of IDEF0 for the Design and Specification of Methodologies.

[7] Flow Chart/Process Flow Diagram [Electronic Resource] – Access Mode: https://www.med.unc.edu/neurosurgery/wp–content/uploads/sites/460/2018/10/Flow–chart–Process–Flow.pdf

[8] Asjad, Sirajuddin. (2019). The RSA Algorithm.

[9] Waykar, Yashwant. (2013). "A Study of Importance of UML diagrams: With Special Reference to Large–sized Projects".

[10] Song, Il–Yeol. (2001). Developing Sequence Diagrams in UML. 10.1007/3–540–45581–7_28.

[11] UML Tutorial for C++ – Windows Platform GDPro 5.0 Chapter 7: Collaboration Diagram The Collaboration Diagram What is a Collaboration Diagram [Electronic Resource] – Access Mode: https://kipdf.com/uml–tutorial–for–c–windows–platform–gdpro–50–numbered–arrows–show–the–movement–o_5aac83951723dd2f0a2a1c28.html

[12] Activity Diagram [Electronic Resource] – Access Mode: https://www.cpe.ku.ac.th/~plw/oop/e_book/ood_with_java_c++_and_uml/ch8.pdf

[13] Database Management – an overview | ScienceDirect Topics. [Electronic Resource] – Access Mode: https://www.sciencedirect.com/topics/computer–science/database–management

[14] What is a Database Schema | Lucidchart. [Electronic Resource] – Access Mode: https://www.lucidchart.com/pages/database–diagram/database–schema

[15] Get started with Kotlin [Electronic Resource] – Access Mode: https://kotlinlang.org/docs/getting–started.html

[16] Android's Kotlin–first approach [Electronic Resource] – Access Mode: https://developer.android.com/kotlin/first

[17] Can any c# application be run on linux [duplicate] [Electronic Resource] – Access Mode: https://stackoverflow.com/questions/32280874/can–any–c–sharp–application–be–run–on linux

[18] What is MYSQL [Electronic Resource] – Access Mode: https://dev.MYSQL.com/doc/refman/8.0/en/what–is–MYSQL.html

[19] Introduction to Redis [Electronic Resource] – Access Mode: https://redis.io/docs/about/

[20] Top Google Play Store Statistics 2022 You Must Know [Electronic Resource] – Access Mode: https://appinventiv.com/blog/google–play–store–statistics

APPENDIX A

A.1 Admin Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.dto.LoginResp;
import com.springtest.demo.dto.ResponseData;
import com.springtest.demo.entity.Admin;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.redisEntity.Token;
import com.springtest.demo.service.AdminService;
import com.springtest.demo.service.TokenService;
import com.springtest.demo.util.Util;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class AdminController {

    @Autowired
    TokenService tokenService;

    @Autowired
    AdminService adminService;

    @PostMapping("/api/token/admin")
    public ResponseData<LoginResp> adminLogin(@RequestBody Admin admin) {
        ResponseData<LoginResp> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {

            var prompt = adminService.login(admin);

            resp.data = new LoginResp();
            resp.data.prompt = prompt;
```

```
            resp.data.isOkay = (prompt.equals(Prompt.success));

            //if successful
            if (prompt.equals(Prompt.success)) {

                String tokenStr = Util.generateToken("admin:" + admin.adminAccount);

                resp.data.token = tokenStr;

                Token token = new Token();
                token.id = admin.adminAccount;
                token.token = tokenStr;

                //store token in redis
                tokenService.saveToken(token);
            }

            return resp;
        }catch (Exception e) {

            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
            return resp;
        }
    }
}
```

A.2 Bill Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.config.ConfigUtil;
import com.springtest.demo.dto.BillRecord;
import com.springtest.demo.dto.Page;
import com.springtest.demo.dto.ResponseData;
import com.springtest.demo.enums.BillType;
import com.springtest.demo.service.MerchantBillAndOverviewService;
import com.springtest.demo.service.UserBillAndOverviewService;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.format.annotation.DateTimeFormat;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestAttribute;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;
import springfox.documentation.annotations.ApiIgnore;

import java.math.BigDecimal;
import java.util.Date;
import java.util.List;

@RestController
public class BillController {

    @Autowired
    MerchantBillAndOverviewService merchantBillAndOverviewService;

    @Autowired
    UserBillAndOverviewService userBillAndOverviewService;


    @GetMapping("/api/bills/user")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    public ResponseData<Page<BillRecord>> getUserBills(@ApiIgnore
@RequestAttribute("userId") int userId,
@RequestParam("pageSize") int pageSize,
@RequestParam("pageNum") int pageNum,
@RequestParam(value = "min", required = false, defaultValue = "0") BigDecimal min,
@RequestParam(value = "max", required = false, defaultValue =
ConfigUtil.MAX_AMOUNT_STR) BigDecimal max,
@RequestParam(value = "start", required = false, defaultValue = "#{new java.util.Date(0)}")
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE) Date start,
@RequestParam(value = "end", required = false, defaultValue = "#{new java.util.Date()}")
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE) Date end,
@RequestParam("billTypes") List<BillType> requestedBillTypes) {

        ResponseData<Page<BillRecord>> resp = new ResponseData<>();

        try {
            resp.data = userBillAndOverviewService.
                    getUserBillRecordByPage(userId, pageSize, pageNum, min, max, start, end,
requestedBillTypes);
            return resp;
```

```
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error!";
            return resp;
        }
    }

    @GetMapping("/api/bills/merchant")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    public ResponseData<Page<BillRecord>> getMerchantBills(@ApiIgnore
@RequestAttribute("userId") int userId,
 @RequestParam("pageSize") int pageSize,
@RequestParam("pageNum") int pageNum,
 @RequestParam(value = "min", required = false,defaultValue = "0") BigDecimal min,
@RequestParam(value = "max", required = false,defaultValue =
ConfigUtil.MAX_AMOUNT_STR) BigDecimal max,
@RequestParam(value = "start", required = false,defaultValue = "#{new java.util.Date(0)}")
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE) Date start,
@RequestParam(value = "end", required = false,defaultValue = "#{new java.util.Date()}")
@DateTimeFormat(iso = DateTimeFormat.ISO.DATE) Date end,
@RequestParam("billTypes") List<BillType> requestedBillTypes) {

        ResponseData<Page<BillRecord>> resp = new ResponseData<>();
        try {
            resp.data = merchantBillAndOverviewService.
                    getMerchantBillRecordByPageWithUserId(userId, pageSize, pageNum, min, max,
start, end, requestedBillTypes);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error!";
            return resp;
        }
    }
}
```

A.3 ExportAndImport Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.dto.ExportOrImport;
```

```
import com.springtest.demo.dto.Page;
import com.springtest.demo.dto.ResponseData;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.enums.UserType;
import com.springtest.demo.service.ExportAndImportService;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import springfox.documentation.annotations.ApiIgnore;

import java.math.BigDecimal;

@RestController
public class ExportAndImportController {

    @Autowired
    ExportAndImportService exportAndImportService;


    @GetMapping("/api/exports/{pageNum}")
    public ResponseData<Page<ExportOrImport>> getAllExports(@PathVariable int pageNum,
    @RequestParam(name = "pageSize", required = false, defaultValue = "10") int pageSize )
    {
        ResponseData<Page<ExportOrImport>> resp = new ResponseData<>();
        try {
            resp.data = exportAndImportService.getAllExports(pageSize, pageNum);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = ResponseData.unknownError;
            return resp;
        }
    }

    @GetMapping("/api/imports/{pageNum}")
    public ResponseData<Page<ExportOrImport>> getAllImports(@PathVariable int pageNum, @RequestParam(name = "pageSize", required = false, defaultValue = "10") int pageSize ) {

        ResponseData<Page<ExportOrImport>> resp = new ResponseData<>();
        try {
            resp.data = exportAndImportService.getAllImports(pageSize, pageNum);
```

```
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = ResponseData.unknownError;
            return resp;
        }
    }


    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    @PostMapping("/api/export/bank")
    public ResponseData<Prompt> export(
@RequestParam UserType userType,
@ApiIgnore @RequestAttribute("userId") int userId,
@RequestParam BigDecimal amount, @RequestParam String paymentPassword) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {
            responseData.data = exportAndImportService.exportToBank(userType, userId,
amount, paymentPassword);
            return responseData;
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }

    }

    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    @PostMapping("/api/import/bank")
    public ResponseData<Prompt> importFromBank(@RequestParam UserType
userType, @ApiIgnore @RequestAttribute("userId") int userId,
                                               @RequestParam BigDecimal amount) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {
            responseData.data = exportAndImportService.importFromBank(userType, userId,
amount);
```

```
            return responseData;
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }

    @PostMapping("/api/export")
    public ResponseData<Prompt> export(
@RequestAttribute("userId") int userId,
@RequestParam BigDecimal amount) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {
            responseData.data = exportAndImportService.exportToMerchant(userId,
amount);
            return responseData;
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }


    @PostMapping("/api/import")
    public ResponseData<Prompt> importFromMerchant(
@RequestAttribute("userId") int userId,
@RequestParam BigDecimal amount) {

        ResponseData<Prompt> responseData = new ResponseData<>();
        try {
            responseData.data = exportAndImportService.exportToUser(userId, amount);
            return responseData;
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }
}
```

A.4 Merchant Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.config.StaticFileConfig;
import com.springtest.demo.dto.*;
import com.springtest.demo.entity.Merchant;
import com.springtest.demo.enums.FileType;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.service.FileService;
import com.springtest.demo.service.MerchantService;
import com.springtest.demo.service.ServiceUtil;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import springfox.documentation.annotations.ApiIgnore;

import javax.mail.MessagingException;
import java.io.File;
import java.io.IOException;
import java.math.BigDecimal;

@RestController
public class MerchantController {

    @Autowired
    MerchantService merchantService;

    @Autowired
    FileService fileService;

    @Autowired
    ServiceUtil serviceUtil;

    @GetMapping("/api/merchant")
    public ResponseData<Merchant> getMerchantByUserId(@RequestParam("userId")
int userId) {

        ResponseData<Merchant> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = merchantService.getMerchantByUserId(userId);
```

```
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;
    }

    @GetMapping("/api/merchant/{id}")
    public ResponseData<Merchant> getMerchantById(@PathVariable("id") int id) {

        ResponseData<Merchant> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = merchantService.getMerchantById(id);
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;
    }

    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    @GetMapping("/api/merchant/self")
    public ResponseData<Merchant> getMerchant(@ApiIgnore
@RequestAttribute("userId") int userId) {
        return getMerchantByUserId(userId);
    }

    @GetMapping("/api/merchant/overview/{merchantId}")
    public ResponseData<OverviewInfo> getMerchantOverview(@PathVariable int
merchantId) {

        ResponseData<OverviewInfo> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = merchantService.getMerchantOverview(merchantId);
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;
    }
```

```
        @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
        })
        @PostMapping("/api/merchant")
        public ResponseData<Prompt> merchantRegister(@ApiIgnore
@RequestAttribute("userId") int userId,

        @RequestParam("companyName") String companyName,
         @RequestParam("licenseNumber") String licenseNumber,
        @RequestParam("licensePhoto") MultipartFile licensePhoto,
         @RequestParam(value = "phoneNumber", required = false, defaultValue = "")
String phoneNumber,
        @RequestParam(value = "email", required = false, defaultValue = "") String email) {

            ResponseData<Prompt> resp = new ResponseData<>();
            File f = null;

            try {

               Merchant merchant = new Merchant();

               merchant.merchantUserId = userId;
               merchant.merchantName = companyName;
               merchant.merchantLicense = licenseNumber;

               try {
                  f = fileService.storeLicenseImage(licensePhoto);
               } catch (IOException e) {
                  e.printStackTrace();
                  resp.data = Prompt.unknownError;
                  return resp;
               }

               merchant.merchantLicensePhoto = StaticFileConfig.toWebUrl(f.getName(),
FileType.license_image);
               merchant.merchantPhoneNumber = phoneNumber;
               merchant.merchantEmail = email;

               resp.data = merchantService.registerMerchant(merchant);

               if (resp.data != null)
                  f.delete();

               return resp;
            } catch (Exception e) {
```

```
            e.printStackTrace();
            if (f != null)
               f.delete();

            resp.data = Prompt.unknownError;
            return resp;
         }
      }

      @GetMapping("/api/merchants/{pageNumber}")
      public ResponseData<Page<UserAndMerchant>> getMerchants(
   @PathVariable int pageNumber,
   @RequestParam(value = "pageSize", required = false, defaultValue = "10") int pageSize)
{
         ResponseData<Page<UserAndMerchant>> resp = new ResponseData<>();
         resp.status = ResponseData.OK;

         try {
            resp.data = merchantService.getMerchants(pageNumber, pageSize);
         } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
         }
         return resp;
      }

      @PutMapping("/api/merchant/state")
      public ResponseData<Prompt> updateUserState(
   @RequestBody ModifyStateRequest obj) {

         ResponseData<Prompt> resp = new ResponseData<>();

         try {
            Merchant merchant = merchantService.getMerchantById(obj.id);

            switch (obj.state) {
               case frozen -> {
                  resp.data = merchantService.freezeMerchant(obj.id);
                  if (resp.data != Prompt.success)
                     return resp;

                  new Thread(() -> {
                     try {
                        serviceUtil.sendFrozenMessage(merchant.merchantEmail,
merchant.merchantPhoneNumber, obj.reasons);
                     } catch (MessagingException e) {
```

```
                    e.printStackTrace();
                }
            }).start();

        }
        case normal -> resp.data = merchantService.unfreezeMerchant(obj.id);
        case unverified -> throw new Exception();
      }

    } catch (Exception e) {
      e.printStackTrace();
      resp.data = Prompt.unknownError;
    }

    return resp;
  }


  @PutMapping("/api/merchant/frozenAmount")
  public ResponseData<Prompt> updateMerchantFrozenAmount(
@RequestBody ModifyFrozenAmount obj) {
    ResponseData<Prompt> resp = new ResponseData<>();

    try {

      if (obj.amount.compareTo(BigDecimal.ZERO) > 0)
        resp.data = merchantService.freezeMerchantBalance(obj.id, obj.amount);
      else
        resp.data = merchantService.unfreezeMerchantBalance(obj.id,
obj.amount.multiply(BigDecimal.valueOf(-1)));


      if (resp.data != Prompt.success)
        return resp;

      try {
        Merchant merchant = merchantService.getMerchantById(obj.id);
        new Thread(() -> {

          try {
            serviceUtil.sendFrozenBalanceMessage(merchant.merchantEmail,
merchant.merchantPhoneNumber, obj.reasons, obj.amount);
          } catch (MessagingException e) {
            e.printStackTrace();
          }
```

```
                }).start();
            } catch (Exception e) {
                e.printStackTrace();
            }


        } catch (Exception e) {
            e.printStackTrace();
            resp.data = Prompt.unknownError;
        }

        return resp;
    }


    @GetMapping("/api/merchants/unverified/{pageNumber}")
    public ResponseData<Page<Merchant>> getUnverifiedMerchants(
  @PathVariable int pageNumber,
   @RequestParam(name = "pageSize", required = false, defaultValue = "10")
int pageSize
    ) {

        ResponseData<Page<Merchant>> resp = new ResponseData<>();
        try {
            resp.data = merchantService.getUnverifiedMerchants(pageSize, pageNumber);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            return resp;
        }
    }

    @PutMapping("/api/merchant/unverified")
    public ResponseData<Prompt> acceptUser(@RequestBody int merchantId) {

        ResponseData<Prompt> resp = new ResponseData<>();

        try {
            resp.data = merchantService.acceptMerchant(merchantId);
            if (resp.data != Prompt.success)
                return resp;
            else {

                Merchant merchant = merchantService.getMerchantById(merchantId);
```

```
                new Thread(() -> {
                    try {
                        serviceUtil.sendAcceptMessage(merchant.merchantEmail,
merchant.merchantPhoneNumber);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }).start();
                return resp;
            }

        } catch (Exception e) {
            e.printStackTrace();
            resp.data = Prompt.unknownError;
            return resp;
        }
    }

    @PostMapping("/api/merchant/reject")
    public ResponseData<Prompt> rejectUser(@RequestBody RejectRequest obj) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {

            Merchant merchant = merchantService.getMerchantById(obj.id);

            responseData.data = merchantService.deleteMerchant(obj.id);
            if (responseData.data != Prompt.success)
                return responseData;

            new Thread(() -> {

                try {
                    serviceUtil.sendRejectMessage(merchant.merchantEmail,
merchant.merchantPhoneNumber, obj.reasons);
                } catch (Exception e) {
                    e.printStackTrace();
                }

            }).start();

            return responseData;

        } catch (Exception e) {
            e.printStackTrace();
```

```
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }
}
```

A.5 Pay Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.businessEntity.PaySemaphore;
import com.springtest.demo.businessEntity.PaySemaphorePool;
import com.springtest.demo.dto.*;
import com.springtest.demo.entity.Merchant;
import com.springtest.demo.entity.Pay;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.redisEntity.SessionPay;
import com.springtest.demo.service.*;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import springfox.documentation.annotations.ApiIgnore;

import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.concurrent.TimeUnit;

@RestController
public class PayController {

    @Autowired
    private PayService payService;

    @Autowired
    private SessionPayService sessionPayService;

    @Autowired
    private PayVerifyService payVerifyService;

    @Autowired
    private MerchantService merchantService;

    @Autowired
    private RefundService refundService;
```

```
        @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
        })
        @PostMapping("/api/pay")
        public ResponseData<PayResp> pay(@ApiIgnore @RequestAttribute("userId") int
userId,
                                         @RequestParam(name = "merchantId") int merchantId,
                                         @RequestParam(name = "amount") BigDecimal amount,
                                         @RequestParam(name = "paymentPassword") String
paymentPassword,
                                         @RequestParam(name = "remarks")String remarks
                                         ) {

            amount = amount.setScale(4, RoundingMode.HALF_UP);
            ResponseData<PayResp> responseData = new ResponseData<>();
            responseData.data = new PayResp();

            try{
                var promptAndPay = payService.pay(userId, merchantId, amount,
paymentPassword, remarks);
                responseData.data.prompt = (Prompt) promptAndPay[0];
                responseData.data.payOverview = PayOverview.fromPay((Pay)
promptAndPay[1]);
            } catch (Exception e) {
                e.printStackTrace();
                responseData.errorPrompt = "Error";
                responseData.status = ResponseData.ERROR;
            }
            return responseData;
        }

        @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
        })
        @PostMapping("/api/payWithConfirm")
        public ResponseData<PayResp> payWithConfirm(@ApiIgnore
@RequestAttribute("userId") int userId,
                                                    @RequestParam(name = "sessionId") int sessionId,
                                                    @RequestParam(name = "paymentPassword") String
paymentPassword,
                                                    @RequestParam(name = "remarks") String remarks) {


            ResponseData<PayResp> responseData = new ResponseData<>();
```

```
            responseData.data = new PayResp();

            try {

                //if the it is not initialized
                PaySemaphore paySemaphore = PaySemaphorePool.getInstance().get(sessionId);
                if (paySemaphore == null) {
                    responseData.data.prompt = Prompt.pay_session_id_error;
                    return responseData;
                }

                //only one mobile phone can scan at one time
                try {
                    var isOkay = paySemaphore.notScanned.tryAcquire(10, TimeUnit.SECONDS);
                    if (!isOkay)
                        throw new InterruptedException();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    responseData.data.prompt = Prompt.multiple_user_pay_error;
                    return responseData;
                }


                //modify the sessionPay state,indicating prepaid
                Prompt prompt = sessionPayService.phoneScan(sessionId, userId, remarks,
paymentPassword);
                if (prompt != Prompt.success) {
                    paySemaphore.notScanned.release();
                    responseData.data.prompt = prompt;
                    return responseData;
                }


                //notify other thread that waiting for user payment
                paySemaphore.isPaid.release();


                //check whether payment  is finished
                try {
                    var isOkay = paySemaphore.isFinished.tryAcquire(1, TimeUnit.MINUTES);
                    if (!isOkay)
                        throw new InterruptedException();
                } catch (InterruptedException e) {
                    e.printStackTrace();

                    //reset back
```

```
                paySemaphore.isPaid.acquire();

                //reset  scan
                paySemaphore.notScanned.release();
                responseData.data.prompt = Prompt.pay_time_out;
                return responseData;
            }


            //finished okay.
            paySemaphore.isFinished.release();

            //reset back
            paySemaphore.isPaid.acquire();

            //reset phone scan
            paySemaphore.notScanned.release();

            responseData.data = paySemaphore.paySynData.payResp;

            return responseData;

        } catch (Exception e) {
            e.printStackTrace();
            responseData.data.prompt = Prompt.unknownError;
            return responseData;
        }
    }


    @PostMapping("/api/payVerify")
    public ResponseData<VerifyPayResp> verifyPay(@RequestBody String
RSAEncryptedBase64String) {


        ResponseData<VerifyPayResp> resp = new ResponseData<>();
        resp.data = new VerifyPayResp();
        resp.data.prompt = Prompt.unknownError;

        try {

            //extract information
            MerchantVerifyInfo verifyInfo =
payVerifyService.extractInfo(RSAEncryptedBase64String);
            if (verifyInfo == null) {
                resp.data.prompt = Prompt.pay_verify_request_format_error;
```

```
            return resp;
        }

        //fetch sessionPay and check whether it exists or not
        SessionPay sessionPay = sessionPayService.getById(verifyInfo.sessionId);
        if (sessionPay == null) {
            resp.data.prompt = Prompt.pay_time_out;
            return resp;
        }

        // get the paySemaphore and check whether it exists or not
        PaySemaphore paySemaphore =
PaySemaphorePool.getInstance().get(sessionPay.sessionId);
        if (paySemaphore == null || paySemaphore.notScanned.availablePermits() == 1) {
            resp.data.prompt = Prompt.pay_time_out;
            return resp;
        }

        //verify the signature
        Prompt prompt = payVerifyService.verify(verifyInfo, sessionPay);
        if (prompt != Prompt.success) {
            resp.data.prompt = prompt;
            return resp;
        }


        //start persistent payment into database
        Object[] promptAndPay = payService.payWithConfirm(sessionPay);
        prompt = (Prompt) promptAndPay[0];
        Pay pay = (Pay) promptAndPay[1];

        if (prompt != Prompt.success) {
            resp.data.prompt = prompt;
            return resp;
        }

        PayResp payResp = new PayResp();
        payResp.prompt = prompt;
        payResp.payOverview = PayOverview.fromPay(pay);

        //remove data in redis
        sessionPayService.delete(sessionPay.sessionId);

        //assign the payment result to paySyn data
        paySemaphore.paySynData.payResp = payResp;
```

```
                //release the semaphore in order to inform phoneScan
                paySemaphore.isFinished.release();

                resp.data.prompt = prompt;
                resp.data.payId = pay.payId;
                return resp;
            } catch (Exception e) {
                e.printStackTrace();
                resp.data.prompt = Prompt.unknownError;
                return resp;
            }
        }


        @PostMapping("/api/sessionPay")
        public ResponseData<SessionPayResp> requestSessionPay(@RequestBody String
RSAEncryptedBase64String) {

            ResponseData<SessionPayResp> resp = new ResponseData<>();
            resp.data = new SessionPayResp();


            try {

                //check the request format
                var sessionRequest = sessionPayService.extractInfo(RSAEncryptedBase64String);
                if (sessionRequest == null) {
                    resp.data.prompt = Prompt.session_pay_request_format_error;
                    return resp;
                }

                //verify the signature
                Prompt prompt = sessionPayService.verifySessionRequest(sessionRequest);
                if (prompt != Prompt.success) {
                    resp.data.prompt = prompt;
                    return resp;
                }


                //initialize
                SessionPay sessionPay = sessionPayService.initialize(sessionRequest.merchantId,
sessionRequest.amount);

                //initialize pay semaphore
                PaySemaphore paySemaphore = new PaySemaphore(sessionPay.sessionId);
                PaySemaphorePool.getInstance().add(paySemaphore);
```

```
            resp.data.sessionId = sessionPay.sessionId;
            resp.data.prompt = Prompt.success;

        } catch (Exception e) {
            e.printStackTrace();
            resp.data.prompt = Prompt.unknownError;
        }

        return resp;
    }

    @GetMapping("/api/payments/{pageNum}")
    public ResponseData<Page<PaymentWithRefund>> getAllPayments(@PathVariable
int pageNum,
                                                @RequestParam(value = "pageSize", required =
false, defaultValue = "10")
                                                        int pageSize
    ) {

        ResponseData<Page<PaymentWithRefund>> resp = new ResponseData<>();

        try {

            resp.data = payService.getAllPays(pageSize, pageNum);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();

            resp.errorPrompt = ResponseData.unknownError;
            resp.status = ResponseData.ERROR;
            return resp;
        }
    }

    @PutMapping("/api/payment/state")
    public ResponseData<Prompt> refundPay(@RequestAttribute("userId") int userId,
@RequestBody int paymentId) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {
            Merchant merchant = merchantService.getMerchantByUserId(userId);
            responseData.data = payService.refundPayWithMerchantId(merchant.merchantId,
paymentId);
```

```
            return responseData;
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }

    @PostMapping("/api/refund")
    public ResponseData<Prompt> refundPayUsingRSA(@RequestBody String
RSAEncryptedBase64String) {

        ResponseData<Prompt> responseData = new ResponseData<>();

        try {

            RefundRequest refundRequest =
refundService.extractInfo(RSAEncryptedBase64String);

            if (refundRequest == null) {
                responseData.data = Prompt.refund_wrong_request_format;
                return responseData;
            }

            Prompt prompt = refundService.validateSignature(refundRequest);

            if (prompt != Prompt.success) {
                responseData.data = prompt;
                return responseData;
            }

            Merchant merchant =
merchantService.getMerchantById(refundRequest.merchantId);

            return refundPay(merchant.merchantUserId, refundRequest.payId);
        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }
    }

}
```

A.6 Transfer Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.dto.Page;
import com.springtest.demo.dto.ResponseData;
import com.springtest.demo.dto.TransferResp;
import com.springtest.demo.entity.Transfer;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.service.TransferService;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import springfox.documentation.annotations.ApiIgnore;

import java.math.BigDecimal;
import java.math.RoundingMode;

@RestController
public class TransferController {

    @Autowired
    TransferService transferService;

    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    @PostMapping("/api/transfer")
    public ResponseData<TransferResp> transfer(@ApiIgnore @RequestAttribute("userId") int
sourceId,
                                               @RequestParam(name = "targetUserId") int targetUserId,
                                               @RequestParam(name = "amount") BigDecimal amount,
                                               @RequestParam(name = "paymentPassword") String
paymentPassword,
                                               @RequestParam(name = "remarks") String remarks) {

        amount = amount.setScale(4, RoundingMode.HALF_UP);
        ResponseData<TransferResp> responseData = new ResponseData<>();
        responseData.data = new TransferResp();

        try{
            var promptAndTransfer = transferService.transfer(sourceId, targetUserId, amount,
paymentPassword, remarks);
            responseData.data.prompt = (Prompt) promptAndTransfer[0];
```

```
            responseData.data.transfer = (Transfer) promptAndTransfer[1];
        } catch (Exception e) {
            e.printStackTrace();
            responseData.errorPrompt = "Error";
            responseData.status = ResponseData.ERROR;
        }
        return responseData;
    }

    @GetMapping("/api/transfers/{pageNum}")
    public ResponseData<Page<Transfer>> getAllTransfers(@PathVariable int pageNum,
                                        @RequestParam(name = "pageSize", required = false,
defaultValue = "10") int pageSize) {
        ResponseData<Page<Transfer>> resp = new ResponseData<>();

        try {

            resp.data = transferService.getAllTransfers(pageNum, pageSize);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = ResponseData.unknownError;
            return resp;
        }
    }
}
```

A.7 User Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.config.StaticFileConfig;
import com.springtest.demo.dto.*;
import com.springtest.demo.entity.User;
import com.springtest.demo.enums.FileType;
import com.springtest.demo.enums.Prompt;
import com.springtest.demo.redisEntity.Token;
import com.springtest.demo.service.FileService;
import com.springtest.demo.service.ServiceUtil;
import com.springtest.demo.service.TokenService;
import com.springtest.demo.service.UserService;
import com.springtest.demo.util.Util;
import io.swagger.annotations.ApiImplicitParam;
import io.swagger.annotations.ApiImplicitParams;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.multipart.MultipartFile;
import springfox.documentation.annotations.ApiIgnore;

import javax.mail.MessagingException;
import java.io.File;
import java.io.IOException;
import java.math.BigDecimal;
import java.util.Map;

@RestController
public class UserController {

    @Autowired
    UserService userService;

    @Autowired
    FileService fileService;

    @Autowired
    TokenService tokenService;

    @Autowired
    ServiceUtil serviceUtill;

    @PostMapping("/api/user")
    public ResponseData userRegister(@RequestParam("phoneNumber") String phoneNumber
            , @RequestParam("firstName") String firstName, @RequestParam("lastName") String
lastName
            , @RequestParam("passportNumber") String passportNumber,
@RequestParam("country") String country
            , @RequestParam("email") String email, @RequestParam("passportPhoto")
MultipartFile passportPhoto
            , @RequestParam("password") String password, @RequestParam("paymentPassword")
String paymentPassword) {

        ResponseData resp = new ResponseData();
        File f = null;

        try {
            try {
                f = fileService.storePassportImage(passportPhoto);
            } catch (IOException e) {
                e.printStackTrace();
                resp.status = ResponseData.ERROR;
```

```
            resp.errorPrompt = "Error! Please Try Again!";
            return resp;
        }

        User user = new User();
        user.setCountry(country);
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setEmail(email);
        user.setPassportNumber(passportNumber);
        user.setPassportPhoto(StaticFileConfig.toWebUrl(f.getName(),
FileType.passport_image));
        user.setPassword(password);
        user.setPaymentPassword(paymentPassword);
        user.setPhoneNumber(phoneNumber);

        userService.register(user);
    } catch (Exception e) {
        e.printStackTrace();
        resp.status = ResponseData.ERROR;
        resp.errorPrompt = "Error!";

        if (f != null)
            f.delete();

        return resp;
    }
    return resp;
  }

  @PostMapping("/api/token/user")
  public ResponseData<LoginResp> userLogin(@RequestBody Map<String, String>
requestData) {

    ResponseData<LoginResp> resp = new ResponseData<>();
    resp.status = ResponseData.OK;

    try {

        String phone = requestData.get("phone");
        String password = requestData.get("password");

        if (phone == null || password == null) {
            resp.errorPrompt = "Error request format";
            resp.status = ResponseData.ERROR;
            return resp;
```

```
        }

        var userAndPrompt = userService.login(phone, password);
        User user = (User) userAndPrompt.get("user");
        Prompt prompt = (Prompt) userAndPrompt.get("prompt");


        resp.data = new LoginResp();
        resp.data.prompt = prompt;
        resp.data.isOkay = (prompt.equals(Prompt.success));

        //if successful
        if (prompt.equals(Prompt.success)) {

            String tokenStr = Util.generateToken("user:" + user.userId);
            resp.data.token = tokenStr;

            Token token = new Token();
            token.id = user.userId.toString();
            token.token = tokenStr;

            tokenService.saveToken(token);
        }
        return resp;
    } catch (Exception e) {

        resp.status = ResponseData.ERROR;
        resp.errorPrompt = "Error";
        return resp;
    }
}

@GetMapping("/api/user/{userid}")
public ResponseData<User> getUserInfo(@PathVariable(name = "userid") int id) {

    ResponseData<User> resp = new ResponseData<>();
    resp.status = ResponseData.OK;

    try {
        resp.data = userService.getUserByUser(id);
    } catch (Exception e) {
        resp.status = ResponseData.ERROR;
        resp.errorPrompt = "Error";
    }
    return resp;
}
```

```
    @ApiImplicitParams({
            @ApiImplicitParam(name = "token", paramType = "header"),
    })
    @GetMapping("/api/user/self")
    public ResponseData<User> getUser(@ApiIgnore @RequestAttribute("userId") int userId) {
        return getUserInfo(userId);
    }


    @GetMapping("/api/user/overview/{userId}")
    public ResponseData<OverviewInfo> getUserOverview(@PathVariable int userId) {
        ResponseData<OverviewInfo> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = userService.getUserOverview(userId);
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;
    }


    @GetMapping("/api/user/search")
    public ResponseData<Page<OverviewInfo>> searchUsers(@RequestParam("keyword")
String keyword,
                                                        @RequestParam("page") int page,
                                                        @RequestParam("pageCount") int pageCount) {

        ResponseData<Page<OverviewInfo>> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = userService.searchUser(keyword, page, pageCount);
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;

    }
```

```
    @GetMapping("/api/users/{pageNumber}")
    public ResponseData<Page<UserAndMerchant>> getUsers(@PathVariable int pageNumber,
                                                        @RequestParam(value = "pageSize", required = false,
defaultValue = "10") int pageSize) {

        ResponseData<Page<UserAndMerchant>> resp = new ResponseData<>();
        resp.status = ResponseData.OK;

        try {
            resp.data = userService.getUsers(pageNumber, pageSize);
        } catch (Exception e) {
            resp.status = ResponseData.ERROR;
            resp.errorPrompt = "Error";
        }
        return resp;
    }

    @PutMapping("/api/user/state")
    public ResponseData<Prompt> updateUserState(@RequestBody ModifyStateRequest obj) {

        ResponseData<Prompt> resp = new ResponseData<>();

        try {
            User user = userService.getUserByAdmin(obj.id);

            switch (obj.state) {
                case frozen -> {
                    resp.data = userService.freezeUser(obj.id);
                    if (resp.data != Prompt.success)
                        return resp;

                    new Thread(() -> {
                        try {
                            serviceUtill.sendFrozenMessage(user.email, user.phoneNumber, obj.reasons);
                        } catch (MessagingException e) {
                            e.printStackTrace();
                        }
                    }).start();

                }
                case normal -> resp.data = userService.unfreezeUser(obj.id);
                case unverified -> throw new Exception();
            }

        } catch (Exception e) {
            e.printStackTrace();
```

```
            resp.data = Prompt.unknownError;
        }

        return resp;
    }

    @PutMapping("/api/user/frozenAmount")
    public ResponseData<Prompt> updateUserFrozenAmount(@RequestBody
ModifyFrozenAmount obj) {

        ResponseData<Prompt> resp = new ResponseData<>();

        try {

            if (obj.amount.compareTo(BigDecimal.ZERO) > 0)
                resp.data = userService.freezeUserBalance(obj.id, obj.amount);
            else
                resp.data = userService.unfreezeUserBalance(obj.id,
obj.amount.multiply(BigDecimal.valueOf(-1)));

            if (resp.data != Prompt.success)
                return resp;

            try {
                User user = userService.getUserByAdmin(obj.id);
                new Thread(() -> {

                    try {
                        serviceUtill.sendFrozenBalanceMessage(user.email, user.phoneNumber,
obj.reasons, obj.amount);
                    } catch (MessagingException e) {
                        e.printStackTrace();
                    }

                }).start();
            } catch (Exception e) {
                e.printStackTrace();
            }


        } catch (Exception e) {
            e.printStackTrace();
            resp.data = Prompt.unknownError;
        }

        return resp;
```

```
    }

    @GetMapping("/api/users/unverified/{pageNum}")
    public ResponseData<Page<User>> getUnverifiedUsers(@RequestParam(name =
"pageSize", required = false, defaultValue = "10")
                                                       int pageSize,
                                                   @PathVariable int pageNum
    ) {

        ResponseData<Page<User>> resp = new ResponseData<>();
        try {
            resp.data = userService.getUnverifiedUsers(pageSize, pageNum);
            return resp;
        } catch (Exception e) {
            e.printStackTrace();
            resp.status = ResponseData.ERROR;
            return resp;
        }
    }


    @PutMapping("/api/user/unverified")
    public ResponseData<Prompt> acceptUser(@RequestBody int userId) {

        ResponseData<Prompt> resp = new ResponseData<>();

        try {
            resp.data = userService.acceptUser(userId);
            if (resp.data != Prompt.success)
                return resp;
            else {

                User user = userService.getUserByAdmin(userId);

                new Thread(() -> {
                    try {
                        serviceUtill.sendAcceptMessage(user.email, user.phoneNumber);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }).start();
                return resp;
            }

        } catch (Exception e) {
            e.printStackTrace();
```

```
            resp.data = Prompt.unknownError;
            return resp;
        }

    }


    @PostMapping("/api/user/reject")
    public ResponseData<Prompt> rejectUser(@RequestBody RejectRequest obj) {

        ResponseData<Prompt> responseData = new ResponseData<>();


        try {

            User user = userService.getUserByAdmin(obj.id);

            responseData.data = userService.deleteUser(obj.id);
            if (responseData.data != Prompt.success)
                return responseData;

            new Thread(() -> {

                try {
                    serviceUtill.sendRejectMessage(user.email, user.phoneNumber, obj.reasons);
                } catch (Exception e) {
                    e.printStackTrace();
                }

            }).start();

            return responseData;

        } catch (Exception e) {
            e.printStackTrace();
            responseData.data = Prompt.unknownError;
            return responseData;
        }

    }

}
```

A.8 Verify Controller

```
package com.springtest.demo.controller;

import com.springtest.demo.dto.ResponseData;
import com.springtest.demo.service.ServiceUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.mail.MessagingException;
import java.util.Map;

@RestController
public class VerifyController {

    @Autowired
    ServiceUtil serviceUtil;

    @PostMapping("/api/verifyCode")
    public ResponseData sendVerifyCode(@RequestBody Map<String,String>
requestData) {

        ResponseData responseData = new ResponseData<>();

        try {
            switch (requestData.get("type")) {
                case "phone" -> serviceUtil.sendPhoneVerifyCode(requestData.get("target"));
                case "email" -> serviceUtil.sendEmailVerifyCode(requestData.get("target"));
                default -> {
                    responseData.errorPrompt = "Error Request Format";
                    responseData.status = ResponseData.ERROR;
                }
            }
        } catch (MessagingException e) {
            e.printStackTrace();
            responseData.status = ResponseData.ERROR;
            responseData.errorPrompt = ("Error! Email address error! Please Check");
        }catch (Exception e) {
            e.printStackTrace();
            responseData.status = ResponseData.ERROR;
            responseData.errorPrompt = ("Error!");
        }

        return responseData;
    }
```

```
        @GetMapping("/api/verifyCode")
        public ResponseData<Boolean> checkVerifyCode(@RequestParam(name =
"type")String type
                ,@RequestParam(name = "target")String target,@RequestParam(name = "code")
String code) {

            ResponseData<Boolean> responseData = new ResponseData<>();
            responseData.data = true;

            try {
               switch (type) {
                  case "phone" -> responseData.data =
serviceUtil.checkPhoneVerifyCode(target,code);
                  case "email" -> responseData.data =
serviceUtil.checkEmailVerifyCode(target,code);
                  default ->  {
                     responseData.status = ResponseData.ERROR;
                     responseData.errorPrompt = ("Error Request Format");
                  }
               }
            }catch (Exception e) {
               e.printStackTrace();
               responseData.status = ResponseData.ERROR;
               responseData.errorPrompt = "ERROR!";
            }

            return responseData;
         }
      }
```