PREPARED BY ANDRII BOBCHUK
SUPERVISED BY MGR.INŻ. WOJCECH DUDZIK

GOFINANCE

A COMPUTER PROGRAMMING-4
PROJECT

Silesian University of Technology
GLIWICE, the 27th of June, 2022

TABLE OF CONTENTS

TOPIC ANALYSIS

Clarification of the subject, class selection, libraries

The topic of my programming project is a basic **Payment System application**.

Program features:

- Multicurrency money transfer between the clients and between the own accounts of each individual client without commission;
- Loans and deposits of certain interest rates which very depending on the selected account currency (i.e. the loan and deposit interest rate for USD is lower than the interest rate for PLN);
- Different account types a client can own. (Depending on purpose or/and currency);
- A possibility for a user to own multiple multicurrency accounts of various types;
- A user can also own accounts of the same types and currencies.
- A possibility for multiple users to own and manage the same account as long as it is a **Debit** account. A **Credit** account cannot be owned by more than one person;
- Admin Console for the staff to manage (decline/accept/ban) users, loans and deposits;

Additionally:

- An app features a graphic user interface implemented with Qt.
- MVC design pattern which make it easy to maintain the codebase and add new features later.
- The app is written in C++ language of the latest standard in order for the program to be able to support most of the topics covered during the laboratories like **regular expressions** for the data validation, **ranges** for an easier and faster loop iteration, **threads** for the background database access and **filesystem.**
- I was not using a ready-made database engine. Instead, I have implemented my own *serialization* keeping the OOP principles in mind. My basic idea was using a JavaScript Object Notation to represent members of my database in a text file. This way it is easy to read and write the information. I am writing the content to a file using the overloaded file output operator. In some cases, I am using an open-source library https://github.com/nlohmann/json for parsing JSON.

- My model consists of 2 actor entities: Client and Staff which inherit their base features from the BaseUser and 2 server entities: DebitAccount and CreditAccount which inherit from BaseAccount.

EXTERNAL SPECIFICATION

User manual, program execution examples/screenshots

A session starts with the user (or a bank employee) having to either sign up or sign in to his profile.

When it comes to registration, a name, surname, email address, login and a password is required. The email is the only field that must be unique. After a staff member approves it, the client may sign in and start creating his first debit or credit accounts.

When creating an account, the user decides it's currency, chooses the account type suitable for his intentions.

When the user is logged in, he can manage his accounts. One user can have multiple accounts (they differ in type and currency) BUT one account can also have multiple owners. For instance, when the two people (i.e. husband and his wife) decide to manage their money together.

A staff member has the ability to either accept or reject a new client, approve or reject all the accounts he may want to open.

In this way a user may send/receive money, earn additional bonuses (Interest rate, when it comes to deposit accounts) when storing funds on account or get some extra money (up to 10.0000 in USD equivalent) on his credit account in return for some interest rate.

IMPORTANT!

1. DEBIT ACCOUNT. In case the user stores more than 0.00 in any currency on his debit account, a system adds a certain interest (rate depends on currency) to his balance **daily**. Moreover, for recalculation of the user's balance a **compound interest** function is used meaning every time the latest current balance is taken as a principal for the function.
2. CREDIT ACCOUNT. A user is given an account with a loan limit of 10,000 in USD equivalent. He is not fined any loan interest as long as <70% of the limit remains on his account. Otherwise, an interest is subtracted daily. If you are left with only 10% of the initial loan credit limit, you get a warning to pay the debt off as soon as possible. Interest is NOT subtracted at this point.

SCREENSHOTS

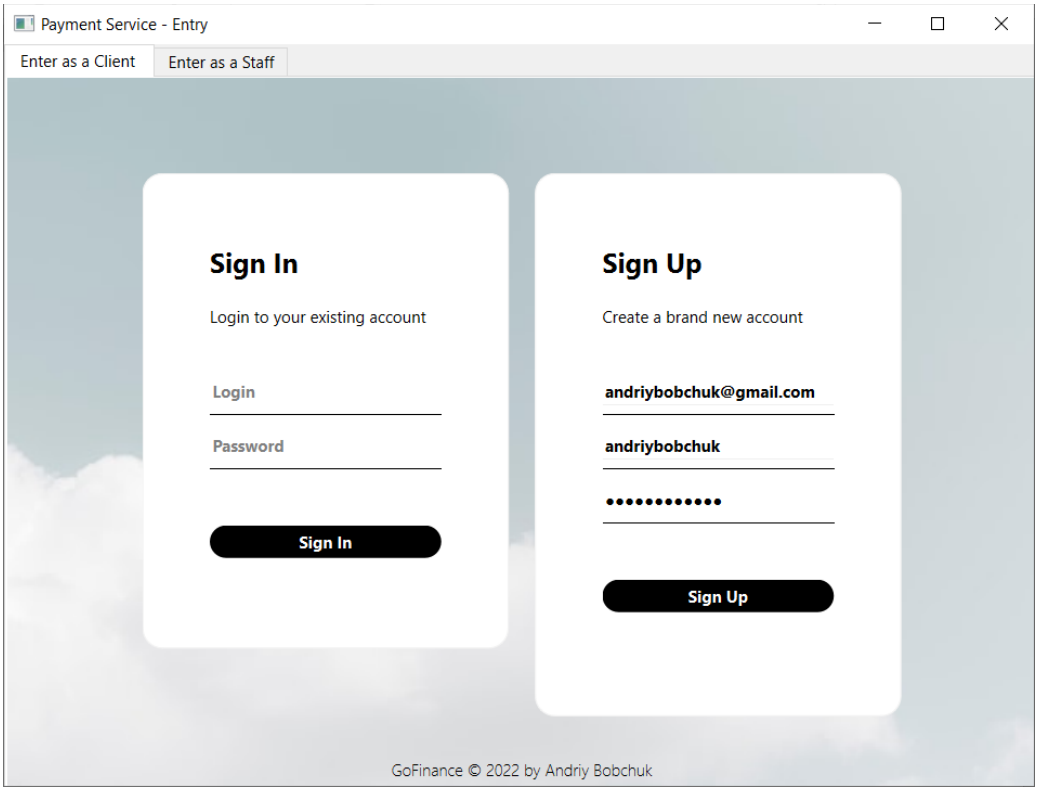


Figure 1 Entry - Client login & registration

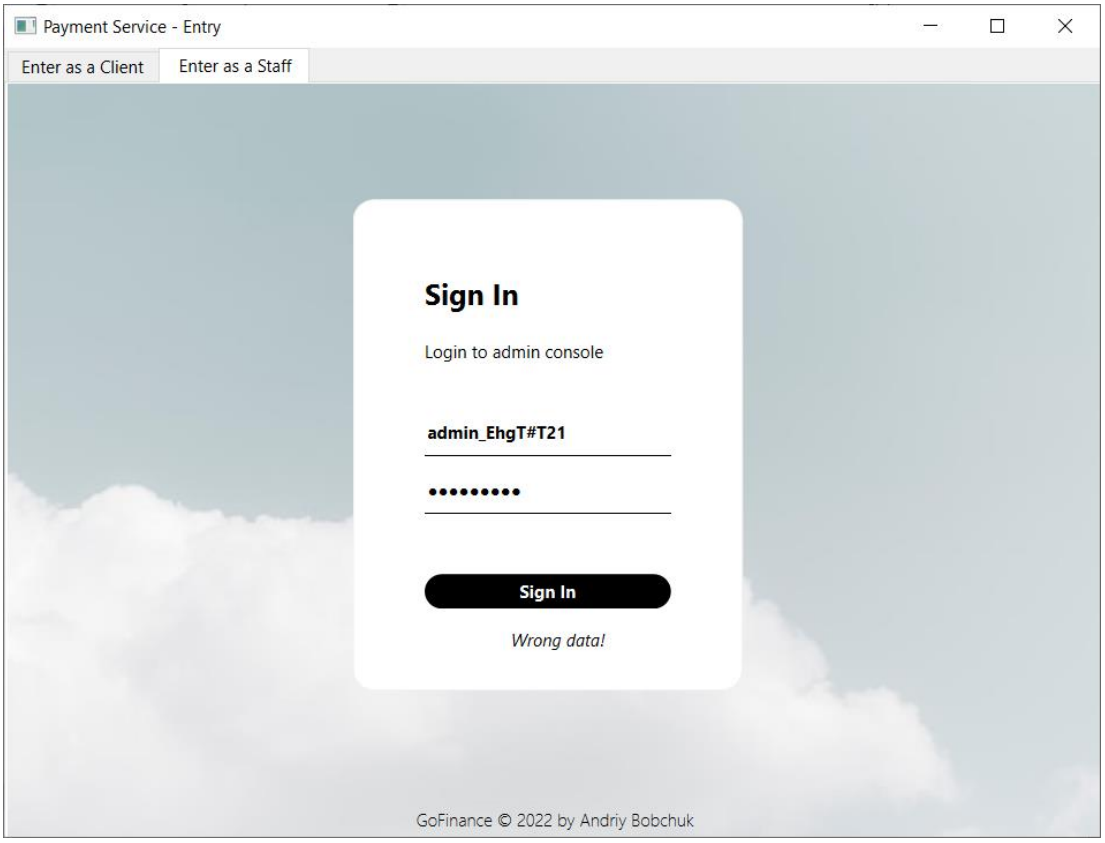


Figure 2 Admin login

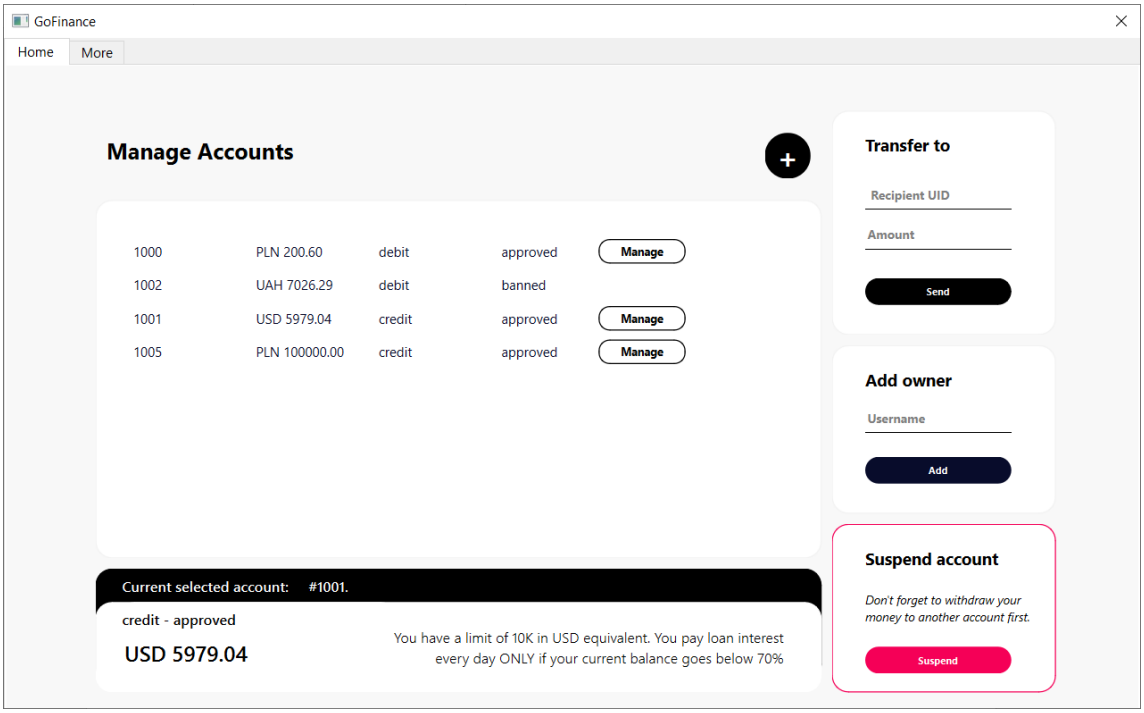


Figure 3 Client dashboard

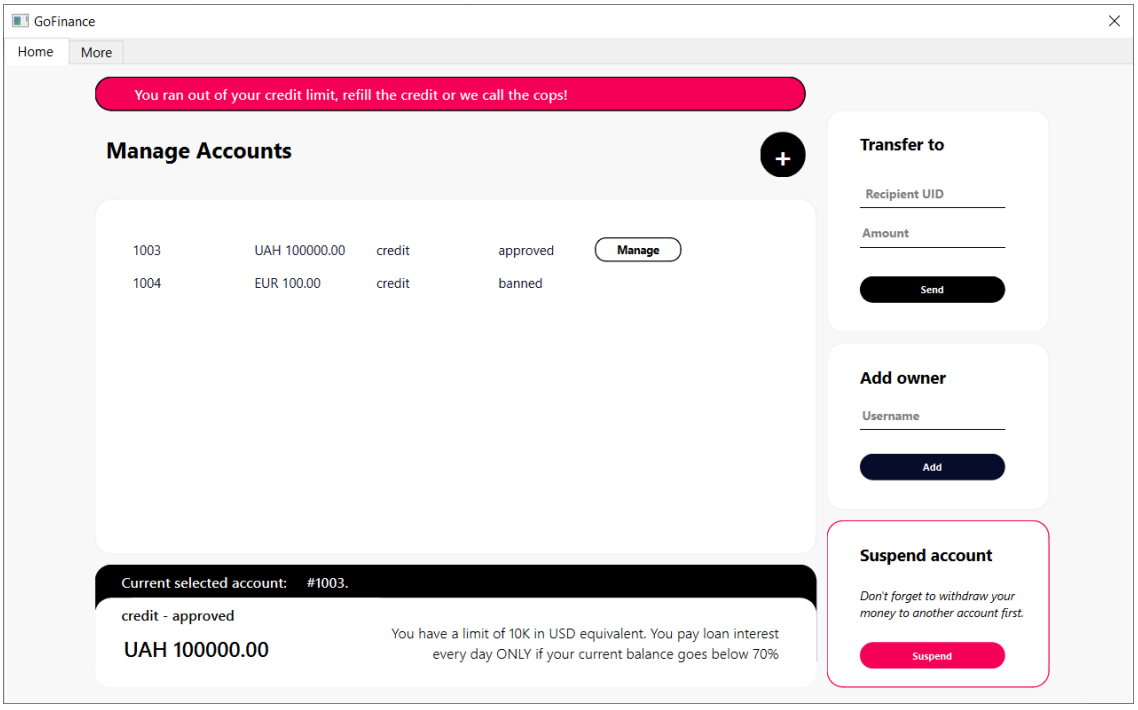


Figure 4 Client dashboard (Notification about unpaid loans)

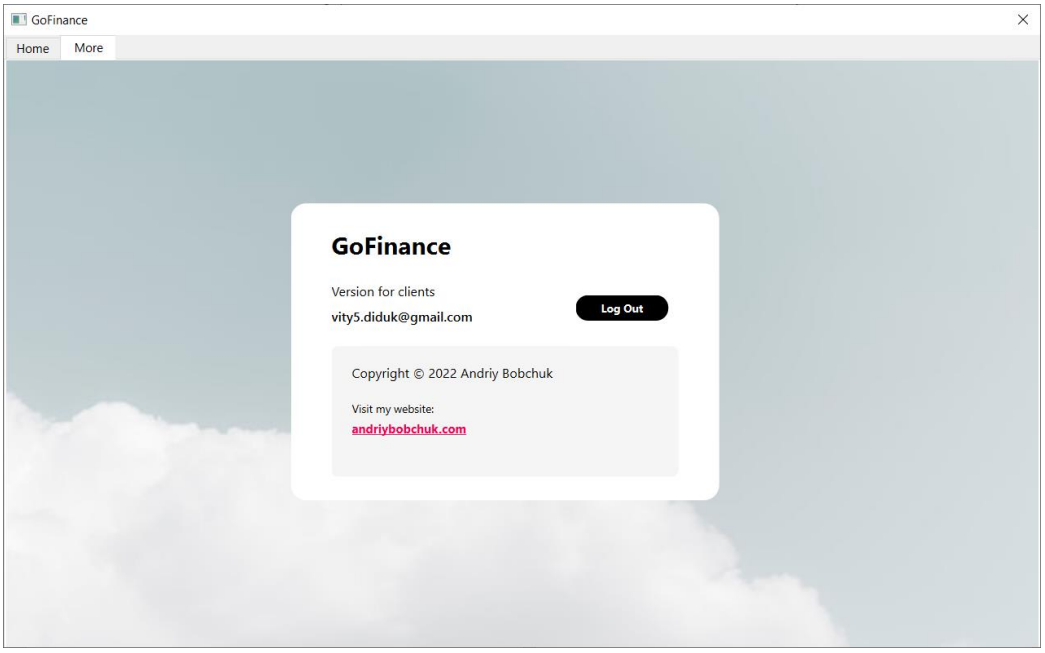


Figure 5 More Tab of Client version

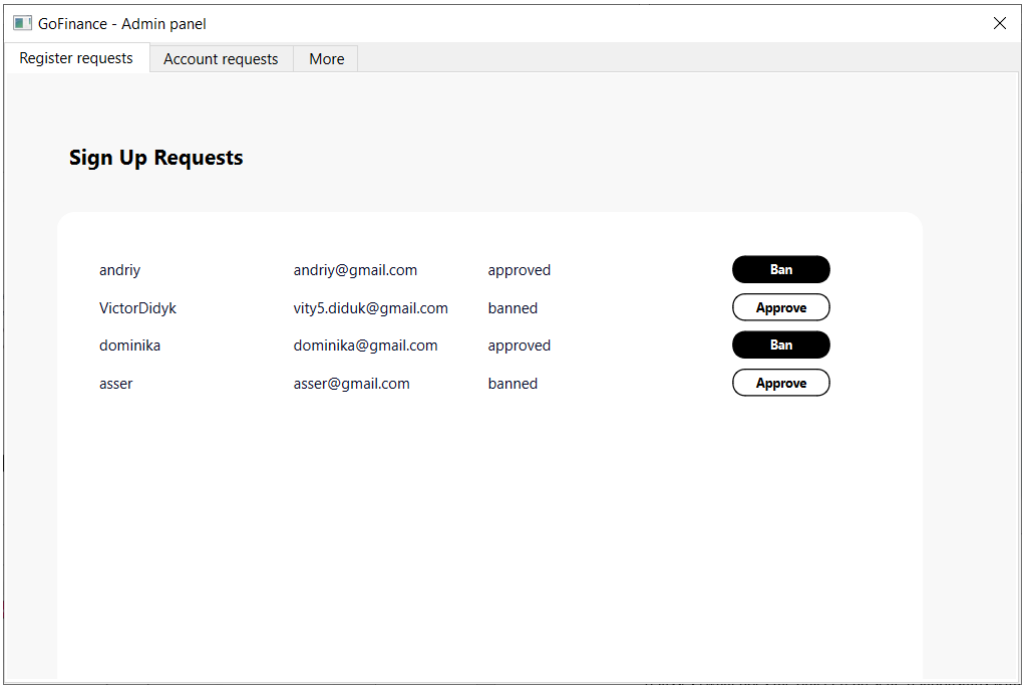


Figure 6 Admin console - Users

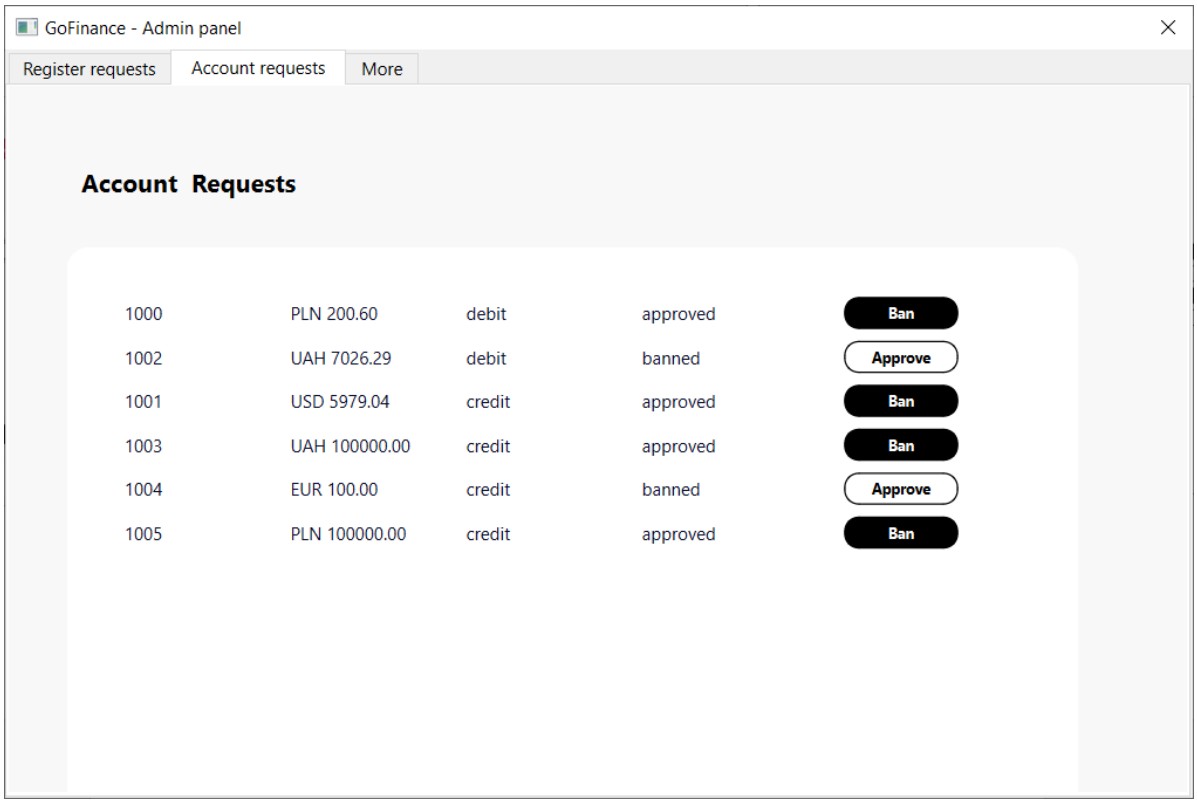
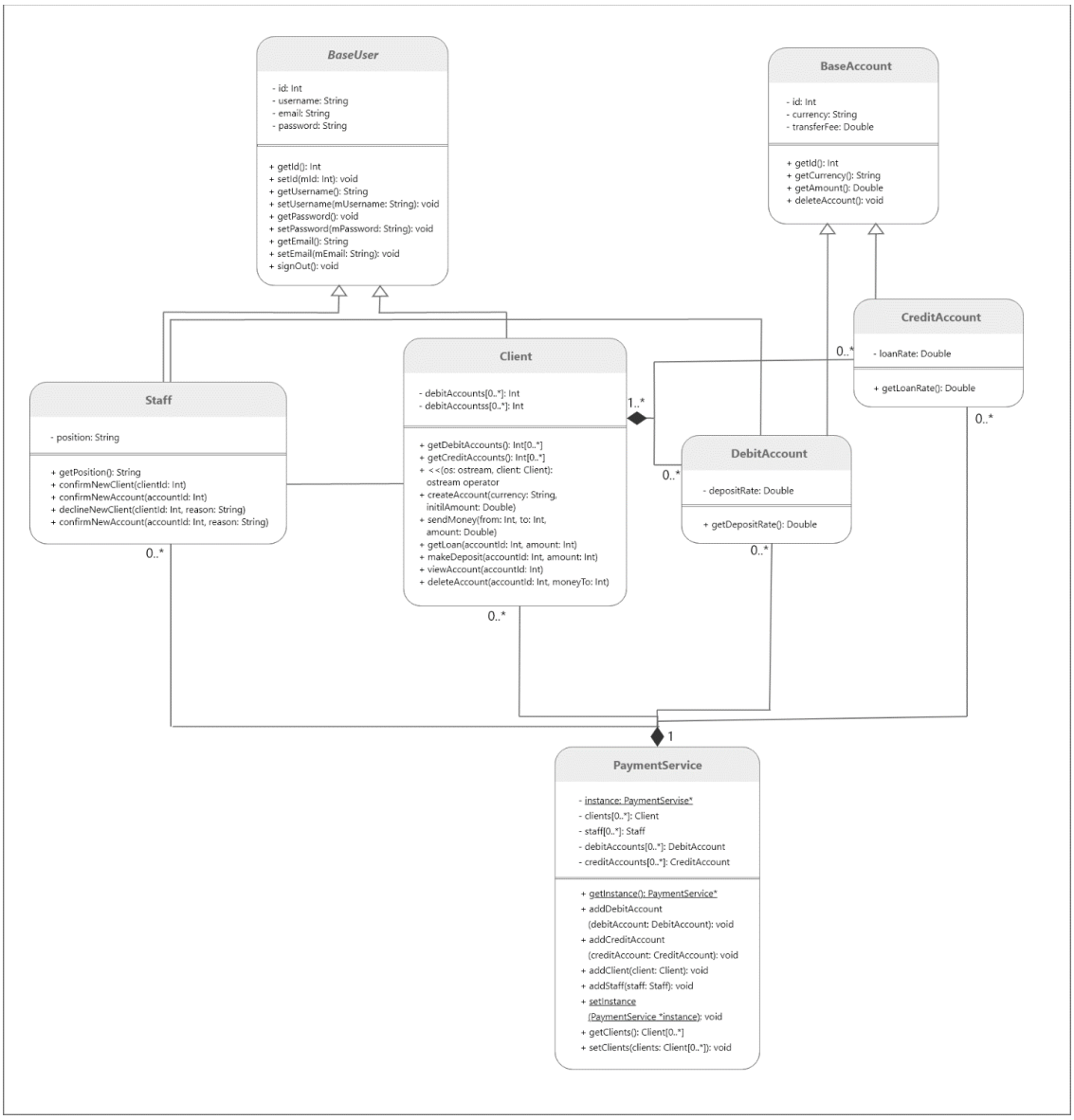


Figure 7 Admin console - Accounts

INTERNAL SPECIFICATION

Classes, class hierarchy diagram, the used object oriented techniques

The following schemes demonstrate the class hierarchy by packages, where Figure 1 represents a model package and Figure 2 – Controller package.



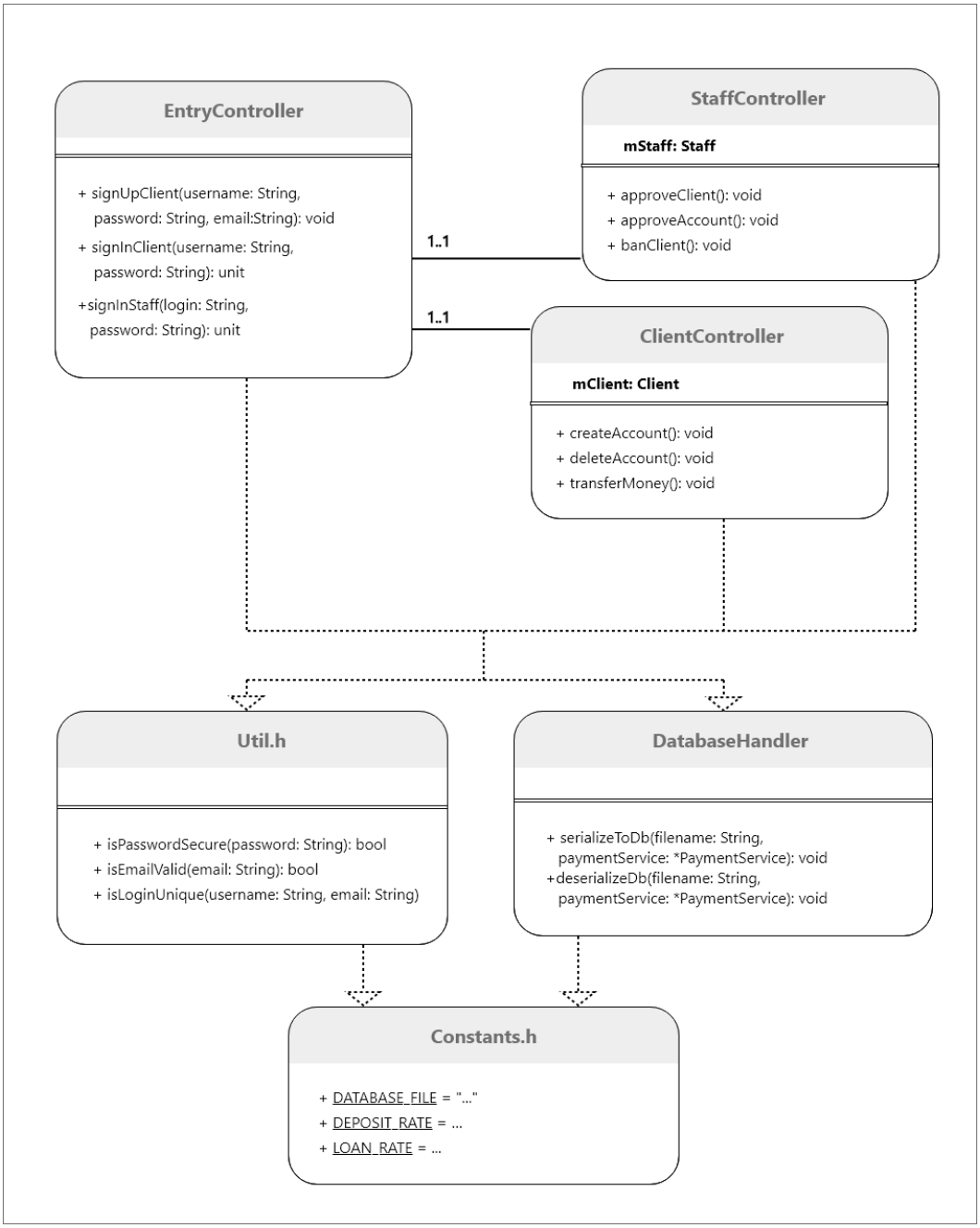


Figure 8 Controller elements together with the dependencies

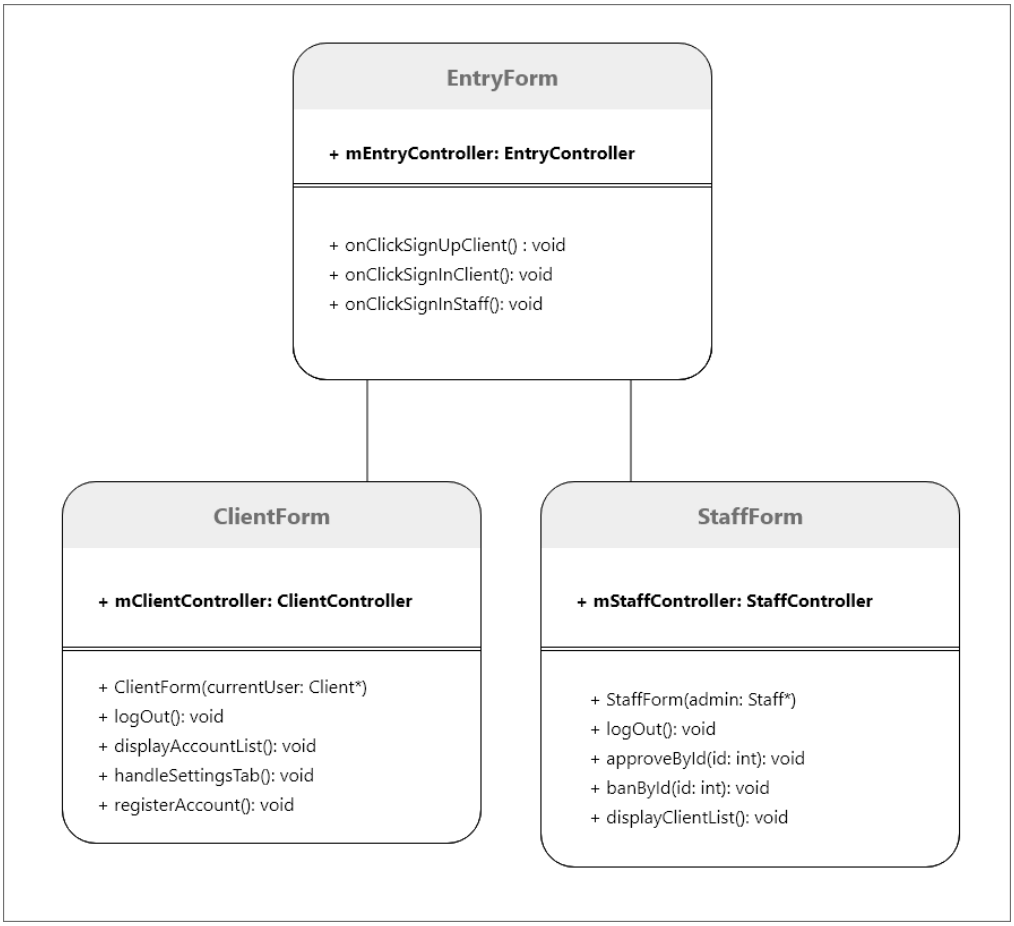
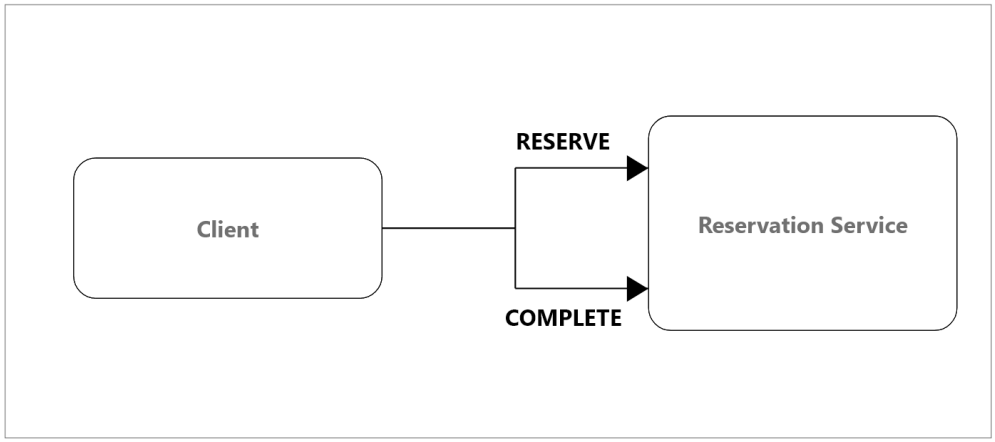


Figure 9 View

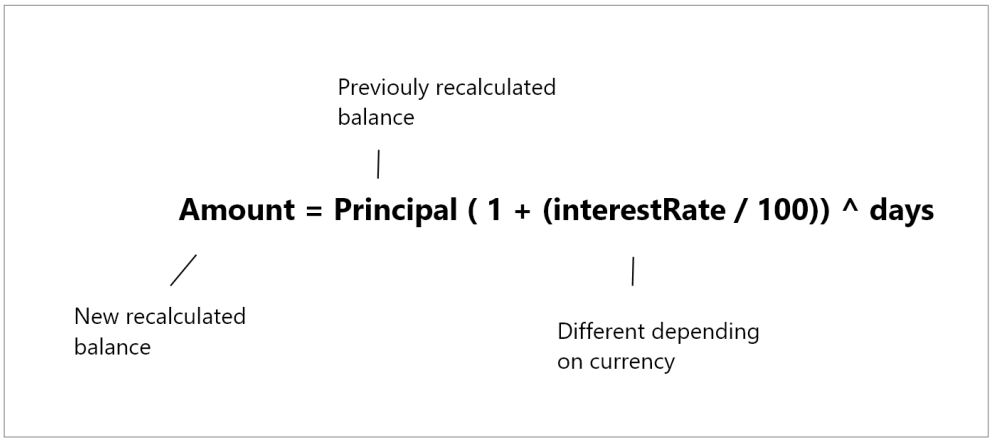
RESERVATION PATTERN

Sometimes it happens that two users are trying to create account at the same time. In this case there is a risk that we obtain two entities with exactly the same property which otherwise must be unique for each client. For this very reason I implemented a reservation class which takes care of this scenario.



COMPOUND INTEREST

In order to calculate the bonuses and penalties for the debit and credit account I decided to use the most realistic method – using the compound interest formula. This way program works like a real bank application.



AVOIDING “MAGIC VALUES” – CONSTANTS.H

In order to avoid hardcoded strings and numbers I created a file with global constants. This way it is easy to quickly change i.e. the name of the database file, loan and/or deposit interest rates, credit limit etc.

```
// Database serialization file
static const std::string DATABASE_FILE = "database.txt";

// Clients' and accounts' statuses
static const std::string PENDING_APPROVAL = "pending approval";
static const std::string APPROVED = "approved";
static const std::string BANNED = "banned";
static const std::string SUSPENDED = "suspended";

// Types of accounts for spinners
static const std::string DEBIT_ACCOUNT = "debit";
static const std::string CREDIT_ACCOUNT = "credit";

// Descriptions for UI based on account types
static const std::string DEBIT_FEATURE = "You receive an interest ra
static const std::string CREDIT_FEATURE = "You have a limit of 10K i
```

Currency exchange rates table

```
// Currency exchange rate table (Relative to USD)
static std::map<std::string, double> RATES = {
    {"USD", 1},
    {"PLN", 0.5},
    {"EUR", 1.1},
    {"UAH", 0.4}
};
```

Base interest rates for deposit and loan

```
// Standard loan/deposit rates in my bank (Per day)
static const double BASE_DEPOSIT_INTEREST = 0.03;
static const double BASE_LOAN_INTEREST = 0.07;
static const double MIN_INTEREST_FREE = 0.7;
static const double CRITICAL_AMOUNT = 0.1;
```

Functions to calculate rates for multiple currencies

```
// Foreign loan/deposit rates
static const double FOREIGN_DEPOSIT_INTEREST(std::string currencyTag) {
    return BASE_DEPOSIT_INTEREST / RATES.find(currencyTag)->second;
}
static const double FOREIGN_LOAN_INTEREST(std::string currencyTag) {
    return BASE_LOAN_INTEREST / RATES.find(currencyTag)->second;
}
```

Base credit limit

```
// Standard loan limit (In USD)
static double BASE_LOAN_LIMIT = 10000;
```

Function to calculate limit for multiple currencies

```
// Calculates loan limit for foreign currencies
static const double FOREIGN_LOAN_LIMIT(std::string currencyTag) {
    return BASE_LOAN_LIMIT / RATES.find(currencyTag)->second;
}
```

OOP TE CHNIQUES AND PATTERNS USED

- Encapsulation: Every class is equipped with all necessary getters & setters
- Inheritance: actor and server subclasses are inheriting base functionality and properties from the superclasses
- Abstraction
- Singleton Design Pattern: runtime database object
- Chain of Responsibility Design Pattern: communication between controllers and views.
- Mediator Design Pattern: PaymentSystem.h

TECHNIQUES COVERED DURING THEMATIC CLASSES

- Regular expressions for the data validation
- Ranges for an easier and faster loop iteration
- Threads for the background database access
- Filesystem.

DATABASE

As I was not allowed to use any ready-made database engine for this project, I decided to implement my own NoSQL-style database which is just a basic file where I store the information expressed as JavaScript Object Notation.

The below-give figure demonstrates the schema of my database.

```
{
   "clients":[
      {
         "username":"andriy",
         "email":"andriy@gmail.com",
         "password":"A195365",
         "status":"approved",
         "debitAccounts":[
            1000,
            1002
         ],
         "creditAccounts":[
            1001,
            1005
         ]
      }
   ],
   "reservedEmails":[

   ],
   "staff":[
      {
         "username":"admin",
         "email":"admin@gmail.com",
         "password":"H8787554",
         "position":"admin"
      }
   ],
   "debitAccounts":[
      {
         "uid":1000,
         "currency":"PLN",
         "amount":200.6,
         "status":"approved",
         "lastRecalculation":"25-06-2022"
      }
   ],
   "creditAccounts":[
      {
         "uid":1001,
         "currency":"USD",
         "amount":5979.04,
         "status":"approved",
         "lastRecalculation":"25-06-2022"
      }
   ]
}
```


Figure 10 Database Schema

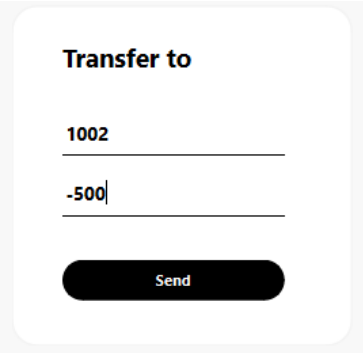
TESTING AND DEBUGGING

During the development I was constantly manually testing the program. Here is the list of some of the detected and fixed bugs:

NEGATIVE TRANSACTIONS

The point of this bug is that a client could choose an account and instead of writing a positive number of the amount they were willing to send to the given account they were stealing any amount they wanted from any existing account.

I fixed this but using REGEX, testing if the number is positive.




```
bool isTransferAmountValid(std::string amount) {

    std::regex validAmountPattern("^[+]?\\d+([.]\\d+)?$");

    if (!std::regex_match(amount, validAmountPattern)) {
        return false;
    }
    else {
        return true;
    }
}
```

ADDING OWNER TO A LOAN ACCOUNT OR DELETING IT

These two bugs were allowing the client to either share the responsibility of the unpaid loan with any other users or just simply get rid of it by closing this account. Now you cannot do neither and you have to deal with your debts by yourself in a fair way.

PERMANENT ACCOUNT DELETION

This one is not a bug but quite a reasonable problem. A user could delete his account and never be able to restore it. I found it bad and decided to delegate admin the ability to restore deleted user accounts. Everything a client should do is just to contact the support.

TRANSACTION RACE

Before program was working in a given way: A user logs in -> program reads the whole database to a runtime object which makes the data access way faster -> when the user logs out, the content of this object is being written to the database. It was causing many errors when multiple users were trying to modify the same account. In order to solve this problem I created two simple functions: pushToDatabase() and pullFromDatabase() which made reading and writing a matter of using one simple function call. From that point I was using these functions to protect the critical regions in my code.