GLO18 Semester project

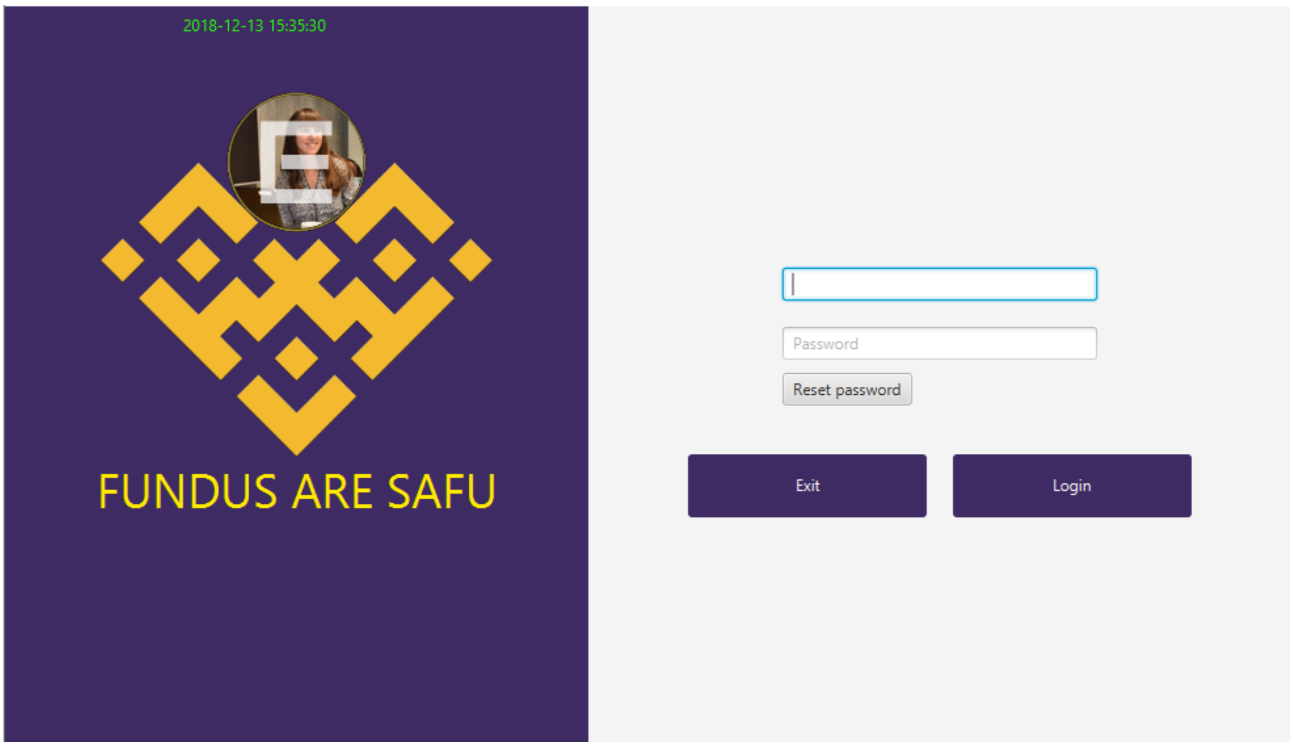
The Bank of Elena

GLO18 Group 10

Group members	Email	Exam number

Supervisor

Institution: University of Southern Denmark
Faculty: Technical faculty
Institute: Mærsk McKinney Møller Institute
Education: BSc in Engineering (Software Engineering)
Semester: 3rd semester
Module code: SB3-GLO-U1
Project period: Week 35-36 to 18-12-2018



The Bank of Elena login screen

I. Abstract

Banking systems are often very complex and require developers to balance user-friendliness with security. This report details the development of a banking system and some of the decisions made during the process. The system consists of three tiers; a client, a web-service, and the database. The major focus of the development has been security, but other design qualities have also been taken into consideration. This report also includes highlights of some aspects of the implementation, such as GUI, database, security measures, and Docker. Testing and debugging measures have also been described. A market analysis has been conducted which looks at the possibility of deploying the system in Nigeria. The report also includes a reflection section, where the group reflects about things they could have done differently. Finally, a conclusion will look at whether or not the project goals were met.

II. Preface

This report is written to show the process and decision making that the group has been through to make a banking system.
The report will cover different topics related to the project, including Background, Problem analysis, Design, Implementation and more.
This report is written with the intention to be used in the GLO semester project exam, and the primary target group for this report is therefore our supervisor and sensor.

III. Table of contents

IV. Editorial

Section of the report	Contributor
Abstract	Peter
Preface	Robin
Introduction	Kim
Background	Peter
Problem analysis	Robin
Theory and methodologies	Jeppe, Peter, Kim, Robin, Frederik, Nick
Requirements	Peter, Robin, Nick, Kim, Frederik, Antonio, Jeppe
Architecture	Nick
Design	Peter, Frederik, Kim, Nick, Jeppe
Implementation	Robin, Antonio, Nick, Kim, Peter
Testing	Robin, Nick
Market analysis	Peter
Reflection	Peter, Nick
Conclusion	Peter
Editing of report structure	Jeppe
Design class diagram	Jeppe

Components of the system	Contributor
Client	Peter, Robin, Nick, Kim, Frederik, Antonio, Jeppe
Web-service	Peter, Robin, Nick, Kim, Frederik, Jeppe
Database	Kim
Docker	Peter
Protocol	Peter, Robin, Nick, Kim, Frederik, Jeppe

1. Introduction

Every day, billions of people use various ways to transfer currency, both locally and globally. With the world getting even more digitalized, the demand for technology increases as well. Today people expect their hard earned money to be safe and available every hour of the day, every day.

While almost everyone has bank accounts and use them indirectly on a daily basis, most have no idea about how they actually work, and how much effort that has been put in to deliver what people expect from such a system.

This semester had a focus on designing a software system in a global context combined with networks and operating systems. This led to a project focusing on solving this complex task by building a secure, functional, and efficient banking system which would be able to communicate across networks. This project would meet all the requirements set by the university, while providing a great opportunity for the project group to learn about banking systems used every day by themselves.

After various research and investigations, the group decided to focus on making the system secure, modifiable etc. But how would the project group ensure this? And would other software qualities be just as important? An example would be how UML and various other descriptions and documents would assist in this process.

This specific banking system would use a custom made protocol to ensure modifiability, and various security related technology would ensure security. But exactly how will this be designed, and later implemented?

To keep track of the process and to make sure there was the required coordination between the group members, various process related tools were used to ensure tasks would be done properly and on time. Which were used and how it went will be explained in this report.

Implementing the system would be no easy task. Besides a proper design and coordination, the skills needed to implement it would require further research and learning by doing. Once the system had been implemented, how would it be tested to ensure it lives up to the requirements? How would the system perform on the global market based on these requirements, for instance in Nigeria.

After the project was finished, various reflections would reveal parts that could be improved, and this project turned out to be no exception. What could have been done better, and did the project end out as hoped and expected, or did it turn out to be a failure? Afterwards, a conclusion will summarize all the results and provide a final overview.

2. Background and motivation

The background and motivation for this project is first and foremost to learn. We chose to make a banking system, because it enables us to incorporate elements from all our courses and beyond.

Banking systems can be both big and complex. It has a plethora of different functions which each need to both be robust and secure. It is therefore essential that the architecture behind the software is made properly. Messy programs are a nightmare to maintain and poor maintenance of software can lead to security flaws. It is also difficult to develop additional features if the current architecture is unstructured.

Along with a good architecture, the project also enables us to work with networks as it is necessary to be able to access the functions anywhere and across all users.

Furthermore, it enables us to discuss different organizational structures in relation to different cultures and thus incorporates cross cultural management. Different cultures have very different approaches to what different employees have access to, and with something as widely used as banking software, it is important to be aware of these differences to make the software useable in as many places as possible.

Beyond these aspects incorporated in our courses, the project also enables us to explore software security. Banks are under constant threat and software faults are potentially detrimental to both bank and customers alike. It is of the utmost importance that the software cannot be breached in any way by unwanted individuals and that internal errors are negligible and discovered quickly. Even small changes can have massive impact on people's and company's finances.

Finally, as banking systems are so complex, it makes it so that we can focus on what we deem as key functions, and then depending on how the project progresses, we can add additional functionality. This gives us more flexibility in the timeline of the project. It is also a good way to practice prioritizing different functions. It is rarely possible to develop all the desired functions of a program, so learning how to prioritize different parts is crucial.

3. Problem analysis

As stated in the background section, the primary goal of the project is to learn how to use the theories and practises from the different courses in the semester. Making a banking system will incorporate a lot of what is taught in the courses this semester and will furthermore fulfil the requirements found in the course description.

When making a banking system, you have to take many stakeholders into account. The stakeholders include the project group, the bank itself and its customers among others. All these stakeholders have different needs and requirements that they want from the system. The requirements have to be defined and ranked so that the system is going to satisfy as many stakeholders as possible.

The question is what do these stakeholders want and what is required of the system? In this day and age, people should not need to physically go to the bank to check the balance or make a transaction. Making a banking system will solve this problem by utilizing a client-server architecture where the customer can sit at home using the client to make a transfer that will be handled by the server instead of physically going to the bank. If a customer wants to open an account, they have to go to the bank to open an account. This is also a problem that a banking system will be able to solve.

The banking system needs to have features that can solve these problems, and some of the features for the banking system could include:

- Accounts for the different users. This could for instance include:
  - An overview of balance(s)
  - A place to transfer money
  - A transaction history
  - An option to create additional bank accounts
- A synchronized service
- Encrypted communication packets
- An encrypted database that stores customer information and transaction history
- The ability for admins to open and close customer accounts

To elaborate on some of the problems that have been identified that the system features should handle:
The system needs to be secure and keep people’s money and information safe and out of reach of hackers and other people who should not have access. This is an obvious requirement when making a banking system, so it is one of the key problems to solve.
The customers should be able to see their account balance, make transactions and see their transaction history, because without these features the system would not be of much use as a banking system. The admins of the system should be able to create and manage the different customer accounts. This feature is needed when a customer contacts the bank and wants to open an account.

To solve the problems the system has to fulfil requirements made from analysing these problems and wanted features. These requirements will be explained and ranked using MoSCoW in section 5 “Requirements”.

3.1 The domain

To further analyse the problem, a domain model has been made to better understand the domain and select the parts of the domain that needs to be part of the system as a minimum. The domain model is shown below in figure 1. The green boxes are the things in the domain that the group has decided to implement in the system and the red boxes are also part of the domain, but with low priority in this project. It was decided that the red parts would only get implemented if time allowed for it.

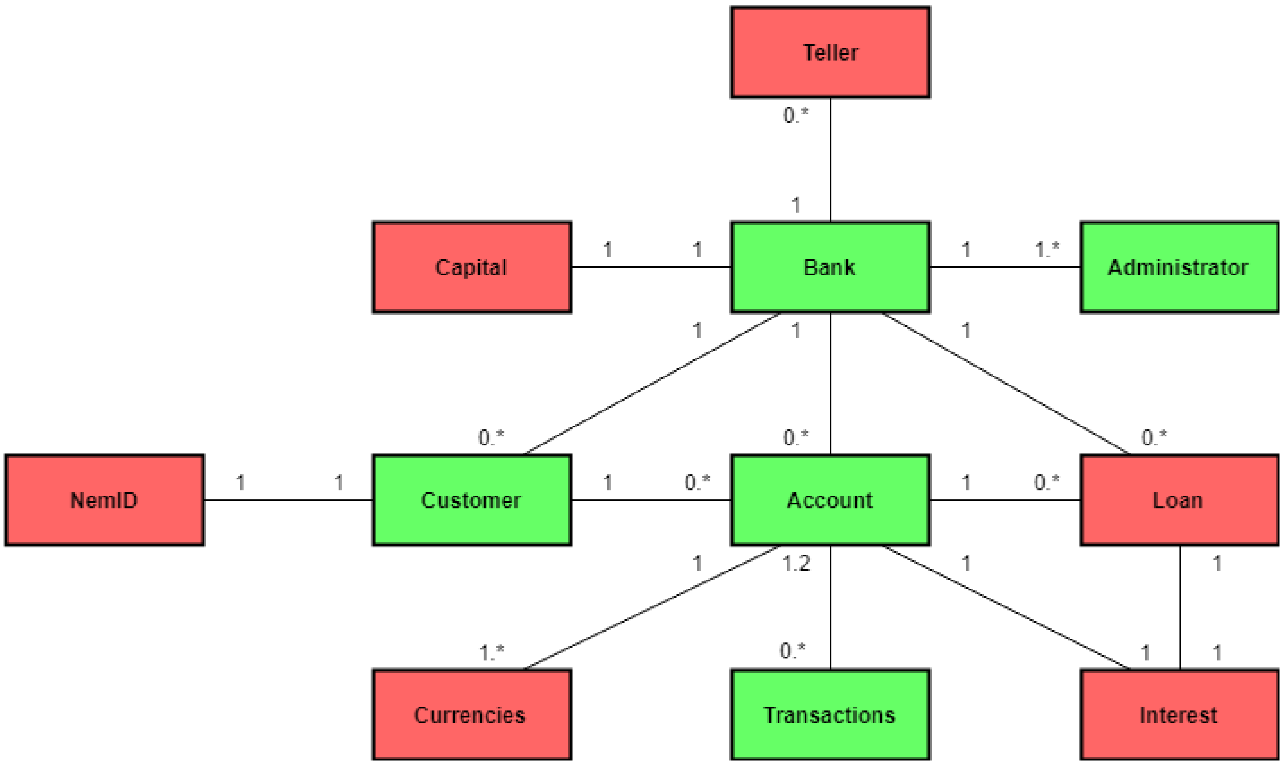


Figure 1 - Domain model

The domain model shows that the domain includes a bank with different people and features connected to it. The bank has customers that have bank accounts and administrators that manages the bank. The bank accounts also include transactions that has been made to or from this account.
The red parts of the model show that there could, for example, be a teller in the bank that can assist the customer if need be. Another part in the domain is NemID which is a secure login with a keycard. This is hard to implement because the group is just university students and not a real bank that could get connected to the nemID system[1].

[1] NemID documentation: https://www.nets.eu/dk-da/kundeservice/nemid-tjenesteudbyder/implementering/Pages/FAQ-for-udviklere.aspx

3.2 Use case overview

A use case overview has been made to give a better overview of what the system should do and how people are going to interact with the system. The diagram shows the different use cases in the system as ellipses and the actors as stickmen who are connected to these use cases with lines between the actors and use cases.

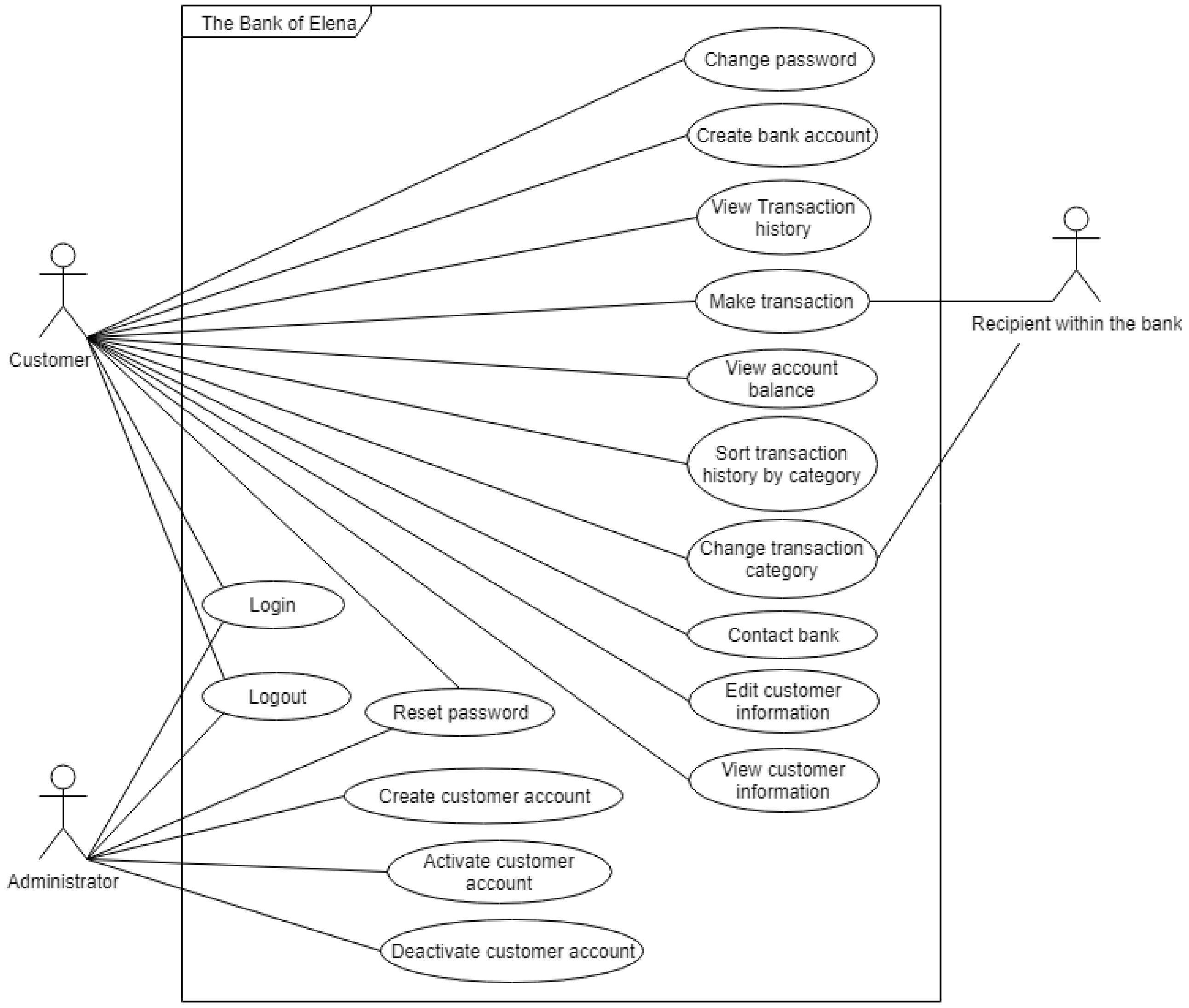


Figure 2 - Use case overview

The customer and admin can both login, logout and reset their password, but other than that, they have different interactions with the system. A customer can for example make a transaction to a recipient within the bank. The category of the transaction can be changed by both people. The sender selects a category that they both can see, but the receiver can change it on their end without changing it on the sender's account history. This way both parties can categorize the transaction as they want.

3.3 Detailed use case descriptions

Some of these use cases have been detailed to more precisely show how the user interacts with the system.

The “login” use case has been detailed and is shown here.

Use case: Login
ID: 1
Brief Description: A user logs into the system
Primary actors: Customer, Admin
Secondary actors: None
Preconditions: The client and web-service programs must be running
Main flow: 1. The use case starts when the user attempts to log in into the system 2. If the customer enters the requested information and presses submit 2.1. If the information entered is valid. 2.1.1. The client makes a server login request 2.1.2. If the request is successful 2.1.2.1. The user is successfully logged into the system 2.1.3. Else an error message is sent informing about the failed request attempt 2.2. Else an error message is displayed and the user is prompted to try again 3. Else an error message is displayed and the user is prompted to fill in the requested info
Postconditions: The system redirects the user to a new screen.
Alternative flows: None

The “Make transaction” use case has also been detailed to show how a transaction is done.

Use case: Make Transaction

ID: 2

Brief description: A customer makes a transaction to a recipient

Primary actors: Customer

Secondary actors: Recipient within the bank

Preconditions: The customer must be logged in to the user account and must have a bank account

Main flow:

```
1.  The use case begins when the customer starts a transaction
2.  The customer selects the account that the money will be transferred from
3.  The customer writes the transaction amount
4.  The customer selects a category
5.  The customer writes the receiving reg. no. and account no.
6.  The customer writes some text for the transaction
7.  The customer submits the transaction
8.  The customer is then asked to confirm the transaction with their password
9.  If information is needed
    9.1.  The customer is asked to fill in the required information
10. Else If the provided information or password is wrong
    10.1.  The customer is asked to write the right information/password
10. If the information in the transaction is correct and there is enough money on the sender's
    bank account
    10.1.  The transaction is sent to the server
        10.1.1.  If the transaction is successful
            10.1.1.1.  The recipient received the money and the customer is informed
                       that the transaction was successful
        10.1.2.  Else if the transaction is unsuccessful
            10.1.2.1.  The customer is alerted about the error
```

Postconditions: The recipient received the money from the transaction

Alternative flows: None

The previous two use cases have been chosen because they show some of the essential functionality in the system. Other use cases from the system has been detailed but have been left out of the report to save space and to spare the reader from reading them all. The other use cases can be seen in appendix C - Detailed use case descriptions.

4. Theory and methodologies

This section will describe the different tools and methods used for the project and the theory and reasoning behind them. These will be sorted into 3 categories; working process, software, and domain. Working process will describe the methods used for the project workflow. Software will describe the methods used for the product. Lastly Domain will describe the general rules and restraints for the domain of a bank.

4.1 Working process

For this project there have been made use of Unified Process (UP) and Scrum. These have been chosen due to their synergy potential, and the group's previous experiences using this combination. The iterative process which UP brings fits perfectly with the sprint oriented Scrum, which allows for easy planning for the different phases of UP.

Scrum

The working process have made use of the agile process Scrum. The parts from Scrum that have been used are the product backlog and sprint backlogs with burndown charts to keep progress of the project. The charts were also valuable tools in the review and retrospect phases to see if the group was over- or underestimating the workload. The banking system is named after the group's supervisor, Elena Markoska, who is the product owner, as well. Scrum was managed in Trello, where the work was distributed among the different team members. The sprint backlog and the product backlog were both managed on Trello. Trello was also used to create a Gantt diagram to manage the time spent on each task. This can be seen in appendix A. Several sprint meetings were held, usually at the start of the day, to catch up on progress and to set the agenda for the day.

Unified Process

The overall work process used unified process as a foundation for how the sprints were designed. Since the group used UP in their last semester, it came naturally and was therefore not documented extensively. The inception phase included a creation of a project foundation, with a focus on stakeholders, requirements, and getting a basic idea of the system. After this the elaboration phase began, the focus here was to analyse the requirements and get an understanding of how the system's architecture and design was going to be. In this phase a prototype was also created to solve the initial problems.

4.2 Software

To make use of the topics learned in "Operating systems and Networking" (OPN), and to make the system as realistic as possible, the group has chosen to make the web-service run in a docker container.
The connection between the client and the server is established with an SSL-socket. This ensures that the connection is secure, and that potential hackers cannot steal plaintext passwords from the packages sent between the client and server.

Docker

Docker is a tool that can create containers to run software in. Containers enable software to be run in an isolated environment, which can be specialised to only have the necessary software and framework for the software to run. In that way, it works much like a virtual machine, however it does not have its own kernel, so it takes up far fewer computing resources than a virtual machine. Furthermore, Docker provides plenty of prebuilt containers through Docker Hub. These containers can easily be modified to be the perfect environment for any software.

Secure Socket Layer / Transport Layer Security and hashing

SSL and TLS are both protocols that enable secure and encrypted communication over the internet. TLS has replaced SSL in most cases and is the go-to algorithm. SSL is often confused with TLS, but they have the same purpose. In this project these names will be used interchangeably as for instance the java implementation uses SSL sockets, which technically use TLS to transfer the data.

In short it works like this:
The client says “hello” and lists different cipher suites, for instance RSA, MD5 and so on. The server says “hello” back and selects a cipher they both have and agree on.

Before they share their keys, a certificate needs to be used to verify the integrity of for instance the server. This means that the client makes sure that the “server” is the actual server it wishes to communicate with. This will deter a middle-man who claims to be the server but in reality, is not. If the client expects a specific certificate and get something else, then something bad might be going on. An example could be when people are browsing and suddenly, they are alerted that a website uses an unsecure or unknown certificate. That is because their browser by default do not trust certificates unless they are verified by external companies as for instance Comodo[2].

Once the certificate is accepted, the client and server share their keys which match, and they can start communicating together.

In this case, symmetric key encryption is used, which means the client and server uses public and private keys which relate to each other. So, the client might use the server’s public key to encrypt some data, which the server can then use its private key to read. And vice versa.

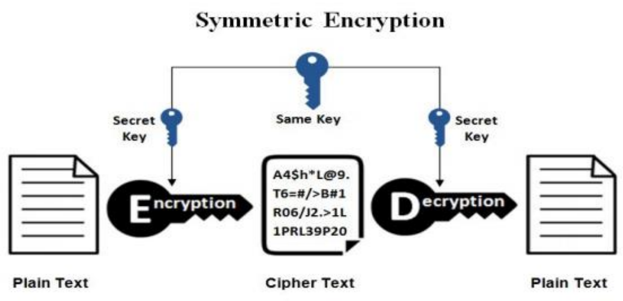


Figure 3 - Symmetric Encryption, taken from OPN lecture 10 slides

[2] Comodo website: https://www.comodoca.com/en-us/?key5sk1=d12644661f97add08ac8a9f2326b3f447d9d5bc1

Hashing

Hashing is also related to SSL/TLS, but the main difference here, is that hashing is irreversible. Encryption, however, can take scrambled text, and reassemble it. It makes sense that TLS uses encryption, since the client or server needs to be able to read the data transmitted, whereas database passwords need to stay secure forever if possible. Hashing in short works by applying mathematical algorithms to make the text unreadable and hard or impossible to reverse. It will be described later how it is implemented, and why it was used. This part was just to make it clear that there is a huge difference between encryption and hashing, and in which contexts they were used. If it was used oppositely, then database passwords would be more insecure, and the client and server wouldn't be able to communicate.

4.3 Domain

There are multiple legal regulations and requirements a bank have to enforce, both on a national level and a global level. The European Union (EU) provides regulations for banks, and then the individual countries can expand on these with further regulations. A recent, and very relevant, addition to that list is the General Data Protection Regulation (GDPR). The GDPR is a set of laws and regulations regarding personal data, and how such data is handled both within the system and in regard to the user.

Laws and regulations

The EU sets multiple requirements to a banking system in form of their regulations for this area. Many of these dictates how the banks should handle money, how money should be managed between accounts, and specific procedures to how information about such is stored. The Bank of Elena should also be subject to all of these regulations. Many of these regulations have been left out due to the time frame for the project and the overall skill level of the group. It has instead been chosen to pick a select few and apply those to the system as proof of concept.
It has been chosen to focus on regulations relevant for storage of user information and activity, along with the regulations relevant for transactions.

From the European Banking Authority (EBA) an interactive single rulebook can be found. This rulebook contains relevant regulations to every aspect of a bank. The rulebook is divided into 5 segments, and of these 5 the *"Payment Services Directive"* section is the most relevant to the system. Within the many articles, regulations on user data and activities can be found.

Taken directly from article 94 in this section:

"Payment service providers shall only access, process and retain personal data necessary for the provision of their payment services, with the explicit consent of the payment service user." [3]

This article shows the general idea for this section of the rulebook, in that minimal amounts of data should be stored about customers, and that customers should be aware and give their consent before this information is to be stored and used. This also complies with other regulations made by the EU, especially the GDPR.

[3] EU rulebook for banks: https://www.eba.europa.eu/regulation-and-policy/single-rulebook/interactive-single-rulebook

General Data Protection Regulation

The GDPR is a set of regulations made to protect European citizens' personal data, and make sure that citizens know what data and information companies gather about them, and how they use it. It also grants citizens the power to request a digital copy of the personal data a company have saved, free of charge.
This idea of public empowerment in regard to digital personal data is further enforced in a citizen's right to be forgotten, meaning that if a citizen wishes to, all their stored information must be deleted, unless this data is required to be saved beyond that by other articles in the GDPR.
The regulations ensure that citizens are not only informed about what data is collected about them when using a company's services, but they also need to give their explicit consent for this data to be collected. It is furthermore a company's responsibility to store gathered information in a secure fashion, and to inform users of breaches as soon as these breaches comes to a company's attention.
The GDPR was adopted in 2016 but did not come into effect until March 2018.

5. Requirements

This part of the report prioritizes the functional and non-functional requirements of system.

Based on the project scope and market research, the immediate ideas for the project have been converted into a list of requirements. These requirements were then prioritized through a discussion of how other banks did their systems and what the most essential functionalities are for the systems. Then the requirements were prioritized using the MoSCoW method and then separated into functional and non-functional requirements. Certain other factors had an impact on the prioritization, these factors are personal skill levels, importance for the system, and the goals for the semester.
The list of requirements has been prioritized with MoSCoW in the following table:

MoSCoW	Functional requirements	Non-functional requirements
Must	1. The client must be able to make requests to the web-service (protocol) 2. The web-service must be able to read/write information from/to the database 3. Store customer information 4. Login to account 5. View account balance 6. Make it possible for a customer to make a transaction to another customer 7. Save all transactions in a transaction history 8. Customers can view their transaction history both incoming and outgoing	● Accurate transactions: 100% of money leaving an account enters another account ● Synchronized system ● A transaction must be completed within 5 minutes ● Availability: The web-service and database must be up at all times during the exam
Should	1. User input must be sanitized 2. Ability to open and close customer accounts 3. Logging of user actions 4. The system should log out after a certain amount of time	● Encrypted communication packets ● Encrypted data in the database

Could	1. Customers can contact the bank 2. Categorize transactions 3. View extra information regarding a transaction 4. Customer can open multiple saving accounts	● Subset of GDPR specifications ● Confirmation before transaction
Would	1. Grant loans to customers 2. Automatically allocate interest to accounts 3. Hierarchy of administrative accounts 4. Store different currencies 5. Convert currencies	● Two-factor authentication ● Reliable (backup database)

The “must” functional requirements are requirements that is considered to be the essential features of a banking system. If these features were not present, it would be difficult to consider the system a functional banking system. A bank where you cannot transfer money from one account to another does not have much purpose. Additionally, the must category includes non-functional requirements such as accuracy and accessibility, so customers do not have to worry about losing money when they use the system.

The requirements under “should” are all related to security. Security is a major concern for all banks. It was therefore decided that after finishing the core functionality, the focus would shift towards securing the system. This is done through both reducing system critical user errors and using encryption to prevent intentional attacks on the system. System critical user errors are errors that either crash the system or change information or money in an unintended way without the user intending to do either. These errors are limited through sanitizing all inputs and deactivating accounts that are no longer customers of the bank. Unfortunately, the automatic log out after a certain amount of time was not implemented, due to time constraints.

Finally, the “could” and “would” categories focus on making the system more user friendly and offering more features for the bank’s customers. Due to time constraints, none of the would features were implemented, but some of the functional requirements in the could category were implemented.

6. Architecture

This chapter will give an overview of both the system architecture and the architecture of individual programs. It will also detail the decisions behind these architectures.

6.1 Overall System Architecture

The architecture that has been chosen is a client-service-database structure, where the client only communicates with the web-service. The web-service communicates with the client and the database. The communication from the client happens when the client sends a message which complies with the protocol, to the webservice. Then the web-service will execute functionality based on a request and send an answer back. The communication to the database happens when the web-service needs some information from the database or needs to update the existing information.

This results in the communication only being initiated one way; the client makes a request to the web-service, which then queries the database. The communication can never be initiated by the web-service or the database.
The Client-Server-Database structure can be seen in figure 4. The client and the web-service also have their own architecture that can be seen in figure 4. Their architecture is nearly identical to each other, the only difference in the architecture, is that the client has a GUI.

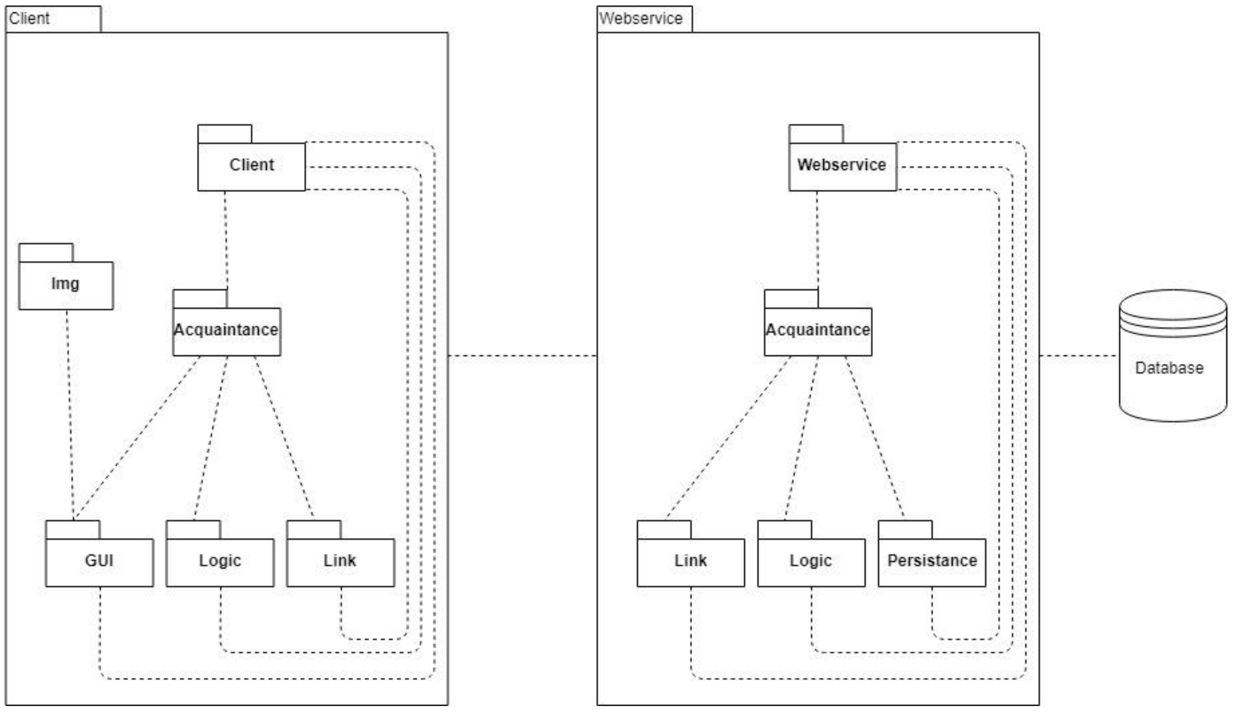


Figure 4 - Architecture overview

6.2 Client Architecture

The architecture in the client has been designed as a 3 layered architecture, with a GUI-, Logic-, Link layer, that each have their own responsibilities, The GUI layer controls the UI, and can call methods in the Logic layer through various interfaces from the acquaintance package.
The GUI gets images from the img package, that it can use in the UI. The logic layer has responsibility for the logic, that is used in the client, and can call the link layer, which sends a message to the webservice.
The communication goes from GUI through Logic to Link, using the interfaces in the acquaintance package to reduce coupling. Lastly, the client package contains the gluecode, which glues it all together at runtime.

6.3 Web-service Architecture

The architecture in the web-service has also been designed as a 3 layered architecture with Link, Logic, and Persistence layer each having their own responsibilities. The Link layer receives a message from the client and sends the message to the Logic layer through the interfaces in the acquaintance package. The logic layer is responsible for checking the message it has received and proceeding according to the message. It can call the Persistence layer, which has the responsibility to communicate with the database, to either update or retrieve information.
The communication goes from Link through Logic to Persistence. The webservice package contains the gluecode, which glues it all together at runtime.

6.4 Database Architecture

The architecture in the database has been setup to contain all the necessary data that needs to be stored for the system to function properly. In the E/R diagram below, it can be seen how the database is structured; both which tables have been created and which attributes the tables contain.
The customer table stores the information of the customers, and has a relationship with bankaccount, a zero to many relationship, because a customer can have zero to many bank accounts, and an account can have 1 to many customers. The transaction, admin and logger tables only store what is needed for them, and they have no direct relationship with the other tables in the database.

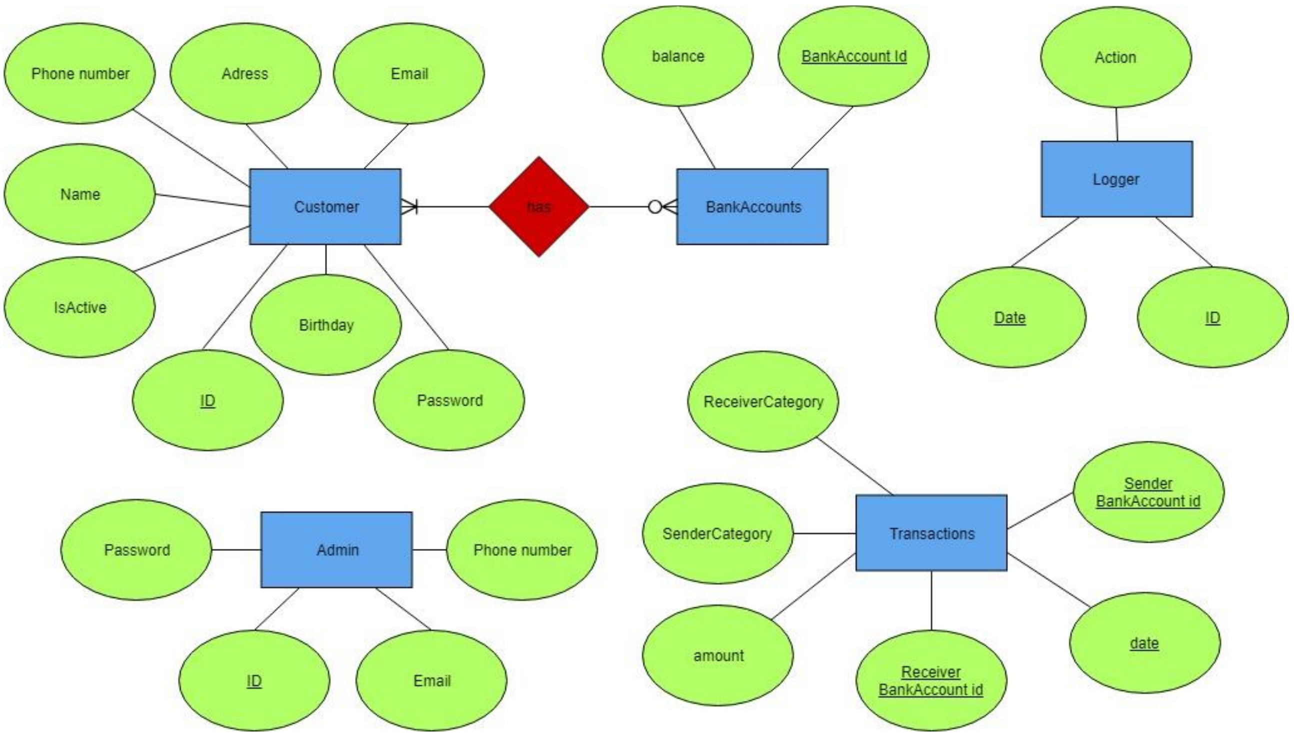


Figure 5 - E/R diagram for the database

6.5 Summary of architecture

To sum up the architecture, the chosen structure is a client-webservice-database architecture. The client communicates with the web-service which is connected to the database. Both applications follow a three-layer structure. For the client, it is GUI-Logic-Link, and for web-service, it is Link-Logic-Persistence. These layers have different tasks. The logic layer is, for example, responsible for all logic in its respective application. Link is responsible for the connection between the two applications. Furthermore, the GUI layer is responsible for the user interface, i.e. what is shown on the screen. The persistence layer is responsible for the database, i.e. storing and reading data.

7. Design

This chapter will describe the main topics surrounding the design choices made for both the individual programs and the combined system. Security has been the main design focus. However, other aspects have also influenced the system.

7.1 Design class diagrams

The system which has been built can at times be very hard to describe with words alone. In order to make the program code easier to comprehend, design class diagrams have been made for the client and web-service. An overview has been created that shows the different packages and their respective classes, and how these are related to each other. Additionally, a detailed view has been created for each of the packages, which each shows fields, methods, and parameters for all the classes and interfaces in their respective packages. Besides an overview, the diagrams also helped the group agree on the design of the system and gain a shared understanding of the system flow. These detailed views can be found in Appendix D.

Client

The client has, as mentioned in the architecture section, been sorted into 3 layers with an acquaintance package to provide interfaces between the layers. The following overview shows the associations between the classes in the different packages and between the packages themselves.

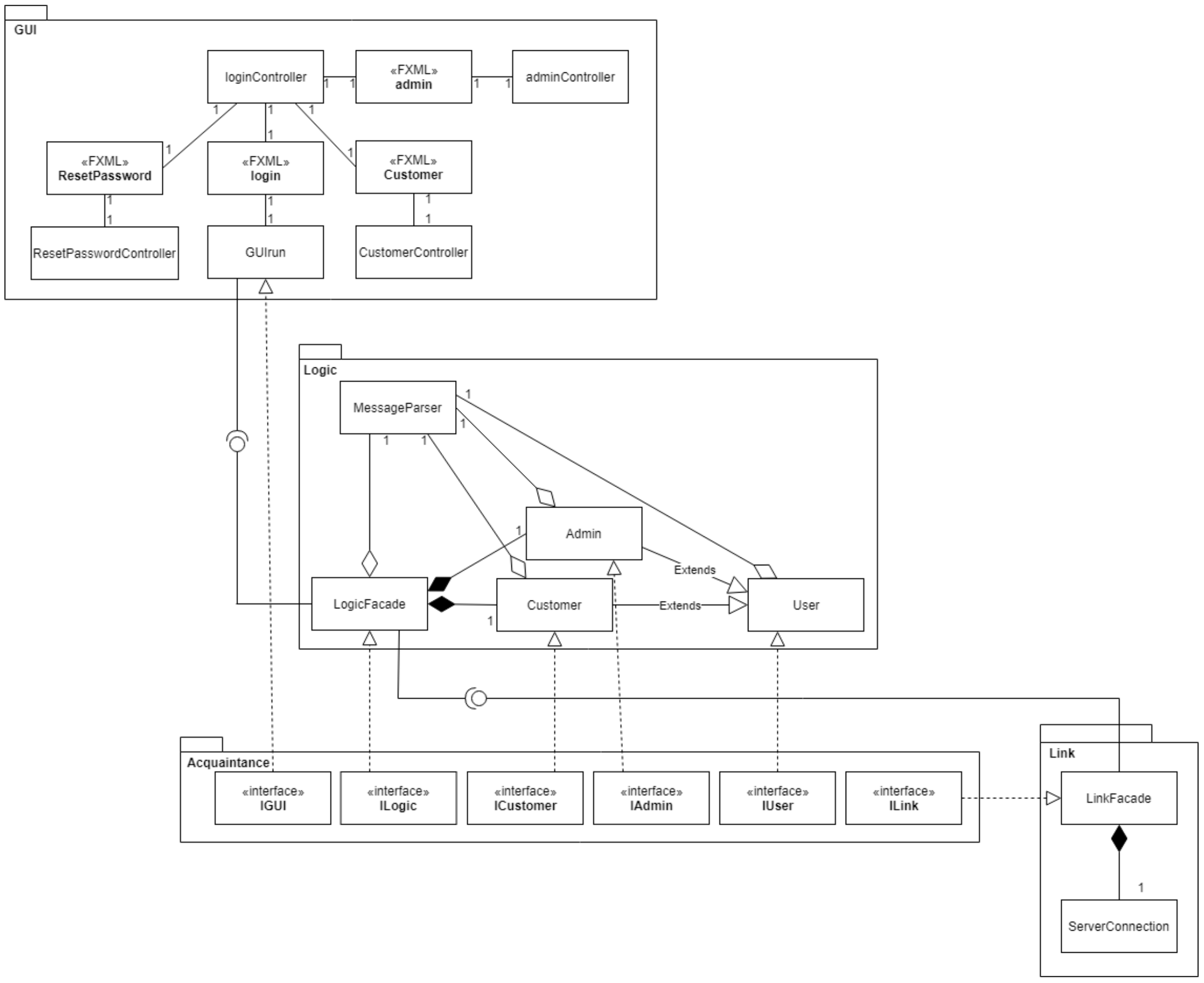


Figure 6 - Design class overview of the client

The overview shows how the FXML files and their respective controllers interact with each other, mainly through scene changes and controller methods. The GUI layer is provided an instance of the LogicFacade through the injection of an ILogic interface, to enable methods from the logic layer to be called from the GUI. Likewise, the logic layer makes use of an instance of LinkFacade through the injection of an ILink interface. The logic layer contains classes to store user information for the user that is currently logged in, be it a customer or an admin. The layer also contains a MessageParser to format messages to the web-service, so they are in line with the protocol. Finally, the link layer handles connection to the web-service, as well as the exchange of messages between the web-service and the client.

Web-service

The server has just as the client been sorted into 3 layers, with an acquaintance package to provide interfaces for communication between layers. The following overview shows the associations between the classes in the different packages and between the packages themselves.

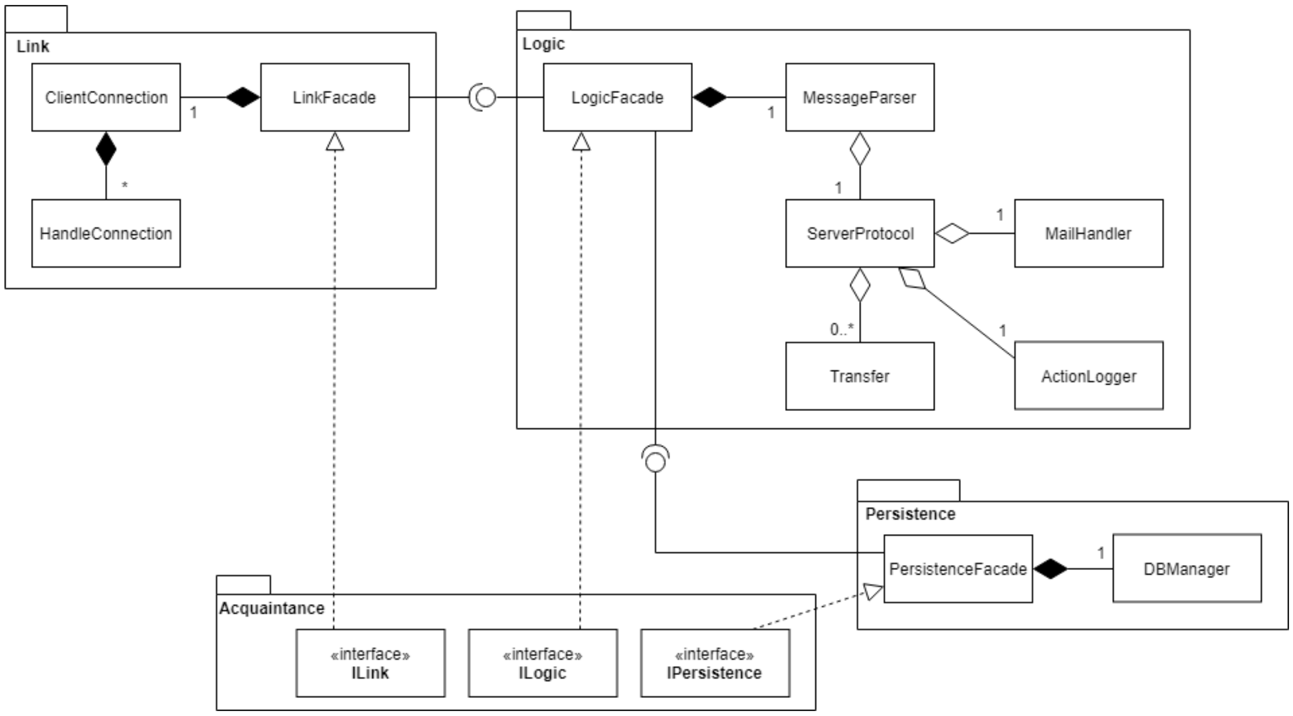


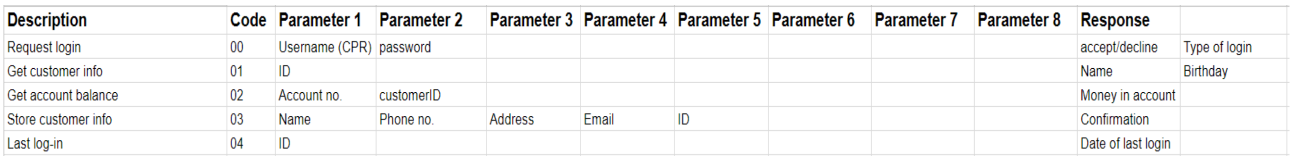
Figure 7 - Design class overview of the web-service

The web-service, just like the client, have a link and a logic layer. The link layer handles client connections and relays messages from the client(s) to the logic layer, by making use of an instance of LogicFacade through the injection of an ILogic interface. The Logicfacade parses the received message and executes functionality based on the defined protocol. This is either done in the logic layer itself, which is the case for mail related functionality, or executes database related methods from an instance of PersistenceFacade, which is an injected implementation of the IPersistence interface. The Persistence layer handles connection to the database and reads or writes information to/from the database.

7.2 Protocol

A major design decision was to figure out a way for the client to communicate with the webservice and vice versa. The project group decided to solve this challenge by implementing a custom made protocol. A protocol is a way to agree on how to communicate to ensure data is sent and received correctly. Instead of choosing a premade one, which could for instance be HTTP the project team now had control of how to customize it to suit the needs of the project.

The protocol was created with simplicity and efficiency in mind, it had to be maintainable and modifiable, and a lot of time was spent on this, as this is a fundamental pillar of the system. The image below shows an example of how the group coordinated the design to ensure the client and web-service would get the expected data.



Description	Code	Parameter 1	Parameter 2	Parameter 3	Parameter 4	Parameter 5	Parameter 6	Parameter 7	Parameter 8	Response	
Request login	00	Username (CPR)	password							accept/decline	Type of login
Get customer info	01	ID								Name	Birthday
Get account balance	02	Account no.	customerID							Money in account	
Store customer info	03	Name	Phone no.	Address	Email	ID				Confirmation	
Last log-in	04	ID								Date of last login	

The protocol works by using “;” (not including the quotation marks) to divide parameters, where Code is the first part of the message. The code is fetched, and a corresponding method is then invoked at the webservice. Each parameter is then part of the invoked method.

A response from the web-service provides knowledge about what the client will receive after its request and is also a help to the developers when coordinating communication. Likewise, maintainability and modifiability were a huge reason for this design. It is easy to add a new protocol, simply add a new code, list the parameters, the response, and start implementing. It made it possible to divide work easily, and once a protocol had been made, two developers could share the same task, where one could implement the client part, while the other could focus on the web-service. As long as the client sent the message as agreed upon per the protocol, it wouldn’t matter how it was implemented in the end.

A downside to this design is that messages sent can’t contain “;” unless it’s perhaps parsed to unicode first, but this is troublesome. It was a trade-off and the group figured that a “;” would rarely if ever be used in banking messages, emails, names, and so on.

When thinking about modifiability, it’s common to imagine what could change, the likelihood of this change, as well as when and by whom this change is made[4]. This correspond with the protocol as it would seem obvious that new functionality would have to be added to the banking system, and in case a component breaks, it is easier to pinpoint where it went wrong due to the custom error messages, for instance “Methodname, error”. Then the developers could take a look at the protocol document and find the area to be fixed. The only downside is that since it’s currently a single class on the web-service handling the initial communication(ServerProtocol) it could grow rather large. The message parser on the client is also prone to grow large. An idea to combat this could be to split it further into modules in the future, or even to include version names along the protocol, in case the webservice and client were running different versions. At the moment different versions would pose an issue as the client and server would not be able to communicate and errors would likely appear.

[4] Lecture slides DES: Quality and Tactics - Modifiability and Availability

7.3 Usability

It is important that the client is user friendly. People need access to their bank accounts often and if the experience is frustrating, it could make them choose another bank. The client has several features to make it easy to use and the GUI is laid out to be as simple as possible. When a customer logs in, they are met by a side-menu on the left side that stays there no matter what they do. This makes it easy for them to navigate to the different features. There has also been made an effort to make tabbing through the input fields and buttons intuitive. This is important as many experienced users prefer to use tab as a way to navigate, because it is often faster when filling out a form. Most of the input fields have the enter key mapped to the same method as the button which is most likely to be pressed next. Many experienced users hit the enter key by reflex after they have filled out an input field. They can easily become frustrated if the enter key does not do what they expect it to do.

Finally, the client displays feedback to the user for all button presses. This is everything from clearing input fields to displaying error messages if something did not work. Good feedback makes the user less likely to make mistakes using the program due to believing that it did not register an action.

7.4 Security

Security is always a fundamental pillar regarding banks. Therefore, this software quality has had a thorough focus throughout the project. The project group did know this from the start, but in order to implement it, it was required to understand exactly what that comprised of.
According to the course book[5] security has three main characteristics; confidentiality, integrity and availability. With that in mind the requirements and structure could be defined.

Security and its impact on usability

Designing for security often conflict with usability, and it was important to find a balanced solution. An example could be the way the system has been designed regarding the sign-up process. The most user-friendly solution would in theory be to have no passwords required, easy sign up for people with barely any security related requirements, and self chosen usernames.
As security was a main focus, it was not made this simple.

Confidentiality

In order to ensure confidentiality, every user was required to have a secure password associated with their account. As accounts were created by a system administrator, it was required to make sure that the administrator would have no information about these passwords. This was made possible by randomly generating secure passwords. The only downside was that the password would be sent to the user over email which is a security risk, but that was a tradeoff to ensure they had access to log in. To compensate for this, they are suggested to change the password upon logging in for the first time.

While a randomly generated password might make logging in more difficult for a user, it ensures that unauthorized people have a much harder time guessing the password. To compensate for the chance that a random password would be really hard to remember, an option to change the password was provided, but with a lower limit of 8 characters.

[5] Book: Software Architecture in practice 3.rd Edition - L. bass, P. Clements, R. Kazman p. 147

Longer passwords are preferred but that limit can easily be changed, and in this project is just supposed to prove a point that a limit is required.

To ensure the stored data was secure, the passwords were hashed using industry approved hashing algorithms with randomly generated salts. The concrete algorithm and such will be described in later parts.

A proper hashing each time will make brute-force attacks slower as generating similar hashes will take time.[6] This impacted usability by delaying the registration process as well as logging in, but tests showed this was almost unnoticeable and therefore a good trade-off. This made sure accounts were protected in case of database leaks. This was the only part of the database which has any sort of encryption or hashing. Encrypting the rest was left out simply to have time for other tasks, and priority was to keep the accounts safe.

Data integrity

Data integrity was ensured as much as time, skill, and money allowed it, by first of all making sure that whatever the client sent was actually what the web-service received, and that no tampering between those two points were possible without a safety measure kicking in.

SSL/TLS and a corresponding safe encryption was used and is an industry accepted standard. Details about this implementation is provided later. The main point is that this standard will ensure data cannot be tampered with between the two endpoints. Once a handshake between the endpoints has been set up, it is very efficient, and delays are barely noticeable according to the various tests the project group made. This also gives confidentiality in case eavesdroppers are monitoring the TCP data packets. A security test was performed as proof of this and will be shown later.

The only downside is that there is no SSL/TLS between the webservice and the database, but this was not taken into account as the project group did not own the database provided by the university, and since the webservice is located on the same network as the database, this was not seen as an issue. The most important thing was to protect a user who might not be aware of potential security flaws.

Prepared statements were also used to ensure integrity and confidentiality. Integrity comes in as SQL injections might tamper with data in queries related to inserting data, for instance passwords, money and such. Confidentiality is relevant during selection queries, only the data supposed to be shown is shown. Exactly how this works will be described later.

While these security measures are implemented to secure the system, it limits the availability a little. If someone else changes the data packages sent between the client and server, TLS will notify the system as per protocol. Availability didn't get affected more than that except if the project group decided to move their frontend to the browser instead of a java based frontend. Because of the SSL/TLS, this would imply that users would have to accept the certificate, that is provided by the backend. This certificate will cost money in order for browsers to approve them by default, and if this was not the case, the website would be rendered as unsafe and less technical people would see the website as not available or malicious.

[6] OWASP iteration count:
https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet#Hash_the_password_as_one_of_several_steps

The java based solution will avoid this issue by making the client automatically accept a certificate that was built into the program. This, however, implies that in case of compromise of the root-certificates that all clients would have to be updated to use the newly issued certificate, and an update would have to be rolled out. Old client versions would not work if the backend suddenly changed certificates, which is a huge problem. But since that was way out of scope for this project, it has not been taken into consideration, what mattered to the project group was to use a free and self signed certificate to make the program available without customers having to accept an unsafe certificate anywhere. The program works out of the box, and therefore a good mixture between availability and security was found.

Balancing various laws and regulations with usability

While the GDPR was considered regarding various logging, another important factor was to consider regulations about what information is visible to the users of the system. An example of this was regarding transactions, as sending money to other people will not reveal any personal information about them. It was considered to show their names as a way to verify the recipient. This turned out to be a potential security flaw, as personal information should stay personal. The project group decided that usability would be deprioritized in this case to keep the users anonymous and as an attempt to live up to the different laws.

Audit and recovery

Finally, there was a focus on the audit part of the security software quality. Audit and recovery go hand in hand and is also helping to make sure the system is available.

This was implemented by logging user actions, users involved, time and so on. This is described in detail later. The main point is that because of this logging, support personnel or possible system administrators will be able to go through the logs and figure out what went wrong in case a customer is, for instance, unable to log in. A log file would in this case perhaps reveal that the customer typed in a wrong password and so on.

In case someone tries to break into the system, that would quickly show in the logs, and countermeasures could start.

All in all, security has been a major focus of this project, and while it might not be perfect, it hopefully shows which software quality that has been a main focus.

7.5 Accuracy

As the system is dealing with money, it is of the utmost importance that the transactions are 100% accurate. The system has some features to ensure accuracy. The main feature which ensures accuracy is the synchronisation of the completeTransfer() method in the Transfer class. The synchronized keyword in Java locks a method if a thread is currently executing it. Another thread wanting to execute the method is then put to sleep until the first thread is done executing the method. Therefore, if two users make a transaction at the same time, only one transaction is processed at a time to ensure that there are no concurrency errors.
However, this insurance only works as long as there is only one instance of the web-service running. If more instances of the software were to be deployed behind a load balancer, other solutions would be needed to keep transactions accurate.

7.6 Scalability

During the development of the Bank of Elena, scalability was one of the qualities which there was less focus on. Because of this and the factors cited in accuracy, the system can only handle one transaction at a time. Therefore, the system is only set up for being vertically scaled. Vertical do have advantages, one being it is more approachable for smaller companies when the demand for a high-speed web-service is not a requirement. Due to the use of threading in the system, using more powerful hardware to get more cores in the CPU, would be the first logical step to scale the system. This becomes a problem if the requirements changes doing development, and desire to scale the system horizontally occurs. Due to design choices in transaction, scaling this way would be a problem. Because of this, reworking transaction could be done to make horizontal scale a possibility.

7.7 Modifiability

This system has been built upon a 3-layer model, which makes it very easy to modify, if anything big needs to be changed. For example, if one day, the need to switch to another kind of database would pop up, that is not based on SQL. The only thing that would need to change in the system, is the persistence layer in the web-service, which could be discarded completely. A new layer could then be built instead.

Another thing, which is easily modifiable, is the protocol that was made, which specifies how the communication between the client and web-service should be formatted. It is easy to add a new rule to the protocol since each request is numbered. Adding a new rule is done simply by adding a new entry and giving it the next number in line.

Another easy thing to modify is the GUI. The way it has been built is that there are 4 scenes; one for login, another for reset password, a third for the admins page, and the last is the one for customers. Things have been separated according to what they are and where they belong. It makes it easier to change things, or add some to the specific part of the GUI, depending on if its the customer that needs more functions, or the admin. This can add redundancy, if some of the things the customer needs, the admin might also need, so one has to add things several places, instead of just one place.

7.8 Interoperability

The only communication between the client and the web-service is done with strings. Due to this, it would be easy to modify the web-service to communicate with any kind of software. Communicating with other software is important if the system were to be deployed in a commercial setting as it is vital to be able to communicate with other banking systems. Furthermore, it would be beneficial to develop a mobile app. An app could be developed without having to change the web-service.

7.9 Summary of design

To summarize the design section, different software qualities are taken into account. The system has gained remarkable results, since all the design choices has been based on these qualities.
A protocol was made to facilitate the communication between the client and the web-service.
Furthermore, security is a fundamental part of the system. By implementing hashing, encrypted communication, randomly generated passwords etc., the likelihood of customers' personal information being leaked has been reduced. Even though security at times conflicts with usability, there has been made an effort to make the client as intuitive as possible. The layered architectures of both the client and the web-service were implemented to ensure that they were both easy to modify. Lastly, the design class diagram gives an overview of the system.

8. Implementation

Some of the most interesting aspects of the system will be highlighted in this chapter. This both includes how parts of the software were coded, but also how the system functions as a whole and the technologies involved in this. This chapter also explains how the design choices were implemented and the impact of them.

8.1 GUI

The GUI of the client program is implemented using Javafx. Javafx uses fxml files to make the GUI and java code as controllers for the GUI and its elements. Fxml files are similar to xml and html in that they are markup languages, so the syntax is similar to some extent. A program called Scene Builder was used to make the fxml files as drag and drop instead of having to write the fxml files manually. Fxml files are used to make scenes, which is just a window with different elements in it.
The program is made to start at the login scene as the first thing it shows the user. If the login is successful, then the scene is changed to another scene. If the user is a customer, then he is shown the customer scene and if he is an admin, he is shown the admin scene.

It is possible to tag different parts of the scene with an fx:id. This could for example be a text field or a button. This tag can then be used in the controller for the scene. This tag will be connected to a variable in the java code. Furthermore, it is possible to assign action handlers to the different parts. These handlers can also be used on text fields, buttons etc.
In the controller for login, there is a handler called login, which is a method that is called when the login button is pressed. In this method it takes the text from the text fields using the fx:id and uses it to check the login info. If the login is correct, the program will change to show another fxml file and use another controller that belongs to this fxml file.
This way it is possible to make a GUI with an fxml file and connect a controller and use it in your program. As stated before, the fxml file and controller is selected when the program starts and then it is possible to transfer control to different controllers depending on which fxml file that is in use. These controllers can then talk to the rest of the system, e.g. the facade of the logic layer or other classes in the GUI layer.

A lot of other things is possible with the use of fxml files and controllers, like making different things visible or invisible/inactive, among other things, in the GUI. These are features that are made possible by setting different values in different variables in the fxml files. These values can be set as a default in the fxml file, but also changed using the controller classes together with the fx:id of the things you want to change.

8.2 System flow

By looking at the interaction diagram for login, the following flow can be observed.

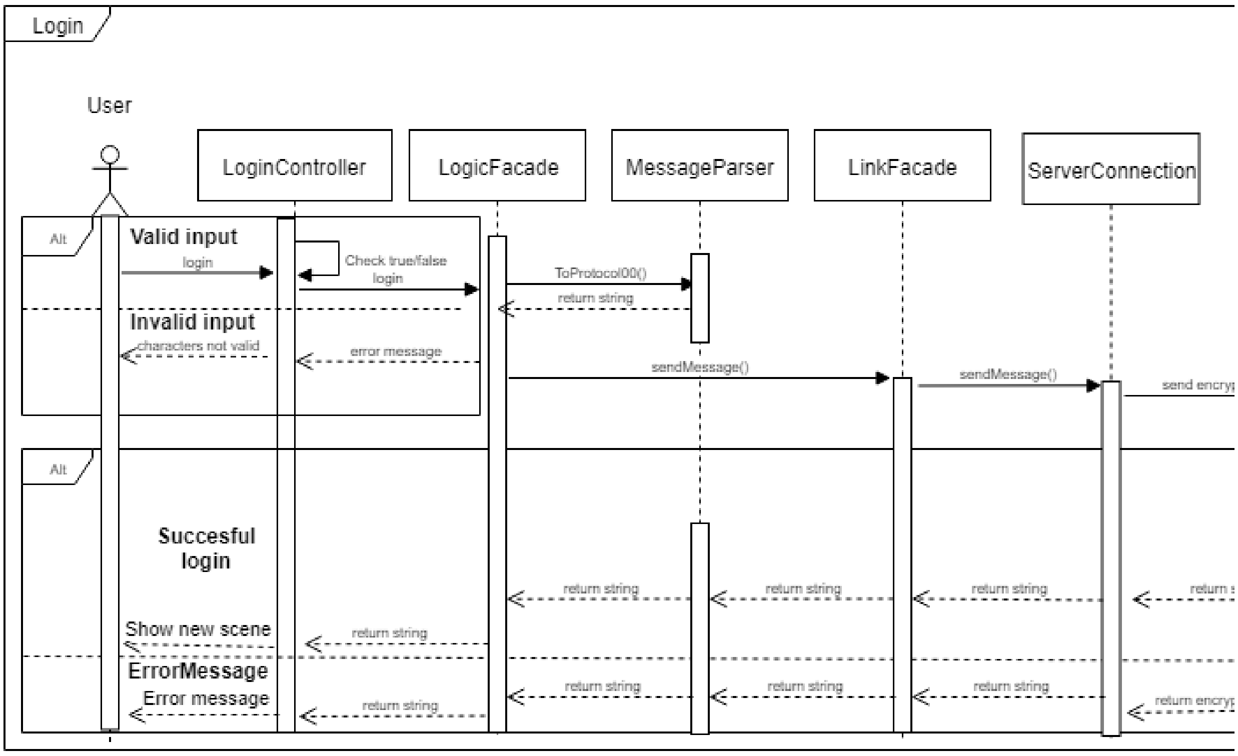


Figure 8 - Interaction diagram for the client part of login

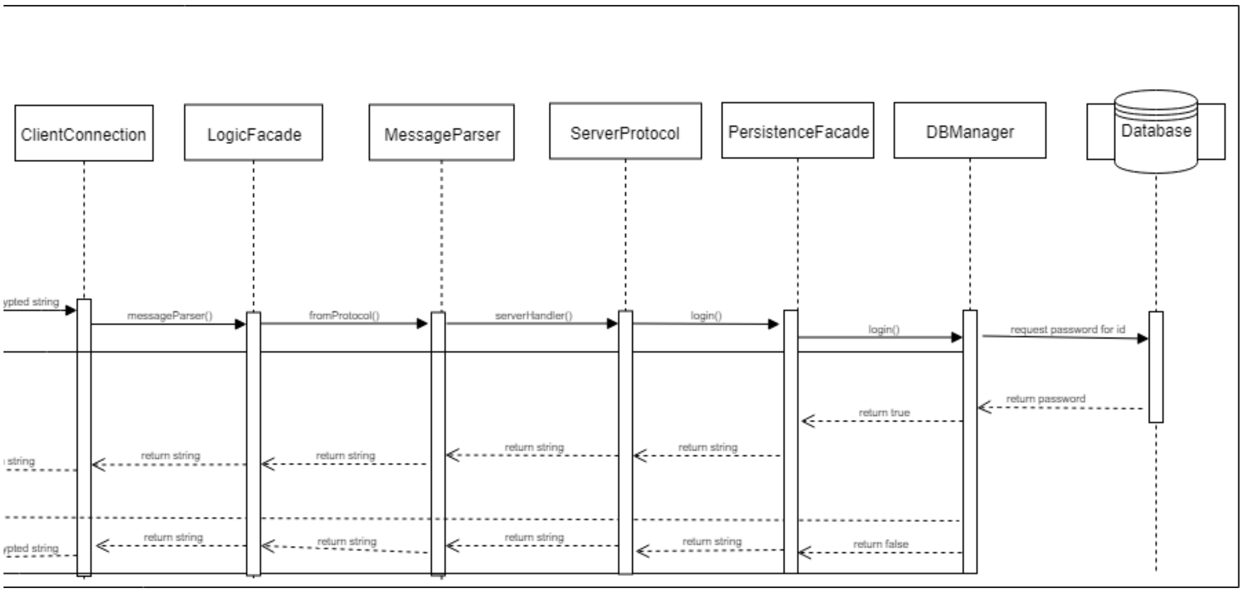


Figure 9 - Interaction diagram for the web-service part of login

After the user has entered the login credentials, a method called login() will first of all validate the input and check if it lives up to the requirements regarding accepted input. Upon success, a method from the logicFacade is called, which then invokes another method called toProtocol00(). This method is written in the messageParser class and will transform the message into something accepted by the protocol mentioned earlier by dividing parameters with ";". Since the message is now ready to be sent, the sendMessage() method is invoked in the logic facade likewise, and will be sent through SSL sockets to the webservice. Upon receiving the message, the message is parsed using messageParser which transforms it into an array, later read by the ServerProtocol.

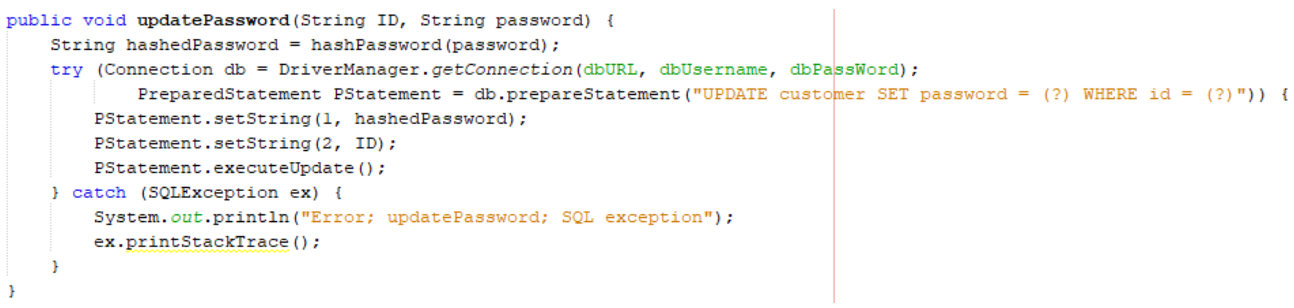
Serverprotocol will depending on the opcode invoke a specific method in a switch statement that further invokes a method from the PersistenceFacade. The PersistenceFacade has access to an object of the DBManager, which contains all the database interaction methods. Upon success or failure, strings will be returned from the DBManager, all the way up to the GUI. Different parts of the system have different error handlers, and that makes it easier to track exactly where it went wrong. By following this pattern, the system is more modifiable, as changing or adding a rule to the protocol won't have any impact on other methods. In case a rule is changed, it would however have to be changed through all layers, which will take extra time, but this trade-off is accepted as the time saved bugfixing is greater than this.

8.3 Database

The database that is being used, is one SDU has made available to the students. It is a postgreSQL database. Each group gets access to their own database with a group login that is used to connect to the database. At the moment, the login is written in clear text in the web-service code, which is not the smartest solution. One reason for this is that currently the source code is publicly accessible on github. Meaning one can simply get the login from there, and this way get access to the database. The group's database has been implemented based on the E/R diagram (figure 5).

The communication to the database happens with prepared statements. Prepared statements are used to protect against SQL injections, an SQL injection is where a user injects an SQL query into the data that is being sent to the database. An example of this, is when a user wants to edit their own information, like their name. They could change it to: "Name; DROP TABLE customer;". What happens then is that the database executes the query to change the users name to Name, and then after that it reads the ; it sees a new query that will then be executed, where that query drops the table with all our customers. This works because it is adding the data directly into the query, where it becomes a part of the query. So, the root of the SQL injection problem is mixing of the code and the data.

The way then that a prepared statement works and how it protects against an SQL injection is that it first sends the query, where the data has been left unspecified with a variable called a placeholder (the ? seen in the figure below on line 4). The database then compiles the query and stores it, without executing it. Then the data will be sent to the database and the query will be executed, with the data sent.
An example of a prepared statement that is being used in the system can be seen in the highlighted code below. This statement updates the user's password. On the third line, it tries to connect to the database, and if that succeeds, it goes to line four where there is the prepared statement with 2 variables ? instead of data. Then on line four and five, we tell it what the data are for the first and second variables in the prepared statement, and then we execute the statement. If the connection would have failed, then on line 8, that would have been caught, and the method would return a sql exception.



```
public void updatePassword(String ID, String password) {
    String hashedPassword = hashPassword(password);
    try (Connection db = DriverManager.getConnection(dbURL, dbUsername, dbPassWord);
            PreparedStatement PStatement = db.prepareStatement("UPDATE customer SET password = (?) WHERE id = (?)")) {
        PStatement.setString(1, hashedPassword);
        PStatement.setString(2, ID);
        PStatement.executeUpdate();
    } catch (SQLException ex) {
        System.out.println("Error; updatePassword; SQL exception");
        ex.printStackTrace();
    }
}
```

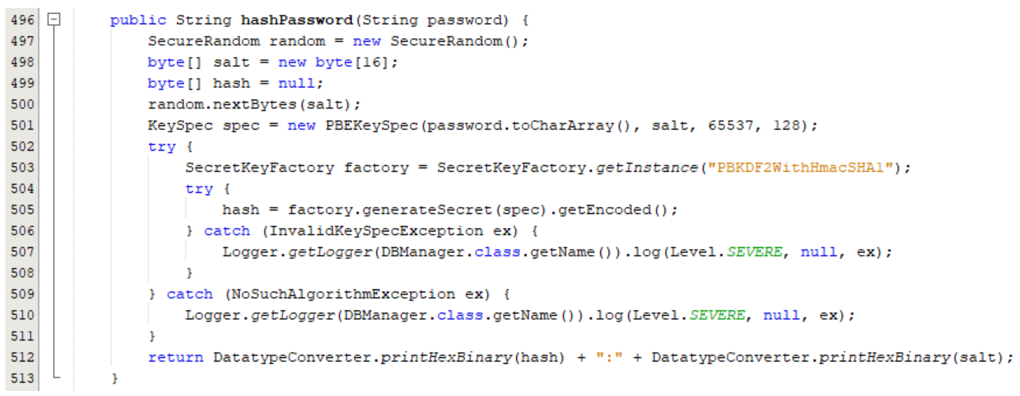
8.4 Hashing

Now that it has been described how to improve security against SQL injections, it is also important to consider the security against anyone with malicious intents who might get access to the database. All an attacker needs is one weak spot, so in order to protect the customers, an extra security measure was put up, called hashing.

The theory behind encryption and hashing has already been described, but in this section, it will be shown how it was implemented and whether or not it succeeded to be secure.

To implement such a functionality the project group relied on industry standards and followed online guides. This approach was justified, as hashing algorithms and such take years and extensive knowledge to make secure. One tiny mistake, and all the data can be cracked, and passwords revealed.

An algorithm named PBKDF2WithHmacSHA1 was used which will, in short, use a securely generated key to hash the password. To assist in making a secure hash, it will furthermore use a randomly generated salt which can be seen as an extra random password in case the user makes a weak password. This will additionally make sure that a possible attacker cannot craft a hashing dictionary beforehand, and simply compare hashes, this is called rainbow tables.[7] This makes sure that even if they get into the database, they would have to start cracking way later than if they were prepared, which gives the bank time to take countermeasures.



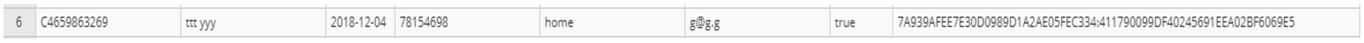
```
    public String hashPassword(String password) {
        SecureRandom random = new SecureRandom();
        byte[] salt = new byte[16];
        byte[] hash = null;
        random.nextBytes(salt);
        KeySpec spec = new PBEKeySpec(password.toCharArray(), salt, 65537, 128);
        try {
            SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
            try {
                hash = factory.generateSecret(spec).getEncoded();
            } catch (InvalidKeySpecException ex) {
                Logger.getLogger(DBManager.class.getName()).log(Level.SEVERE, null, ex);
            }
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(DBManager.class.getName()).log(Level.SEVERE, null, ex);
        }
        return DatatypeConverter.printHexBinary(hash) + ":" + DatatypeConverter.printHexBinary(salt);
    }
```

The iteration count is how many times to apply the final hashing algorithm, for instance SHA1 to the already created hash. The more iterations, the slower it will be to crack these passwords. As it will take more time.[8]

[7] Rainbow tables: https://en.wikipedia.org/wiki/Rainbow_table
[8] Password hashing: https://www.sjoerdlangkemper.nl/2016/05/25/iterative-password-hashing/

Since the salt is randomly generated each time, how can the users log back in?

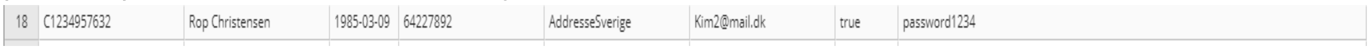
This problem was solved by appending the salt next to the hashed password in the database, as so.



6	C4659863269	ttt yyy	2018-12-04	78154698	home	g@g.g	true	7A939AFEE7E30D0989D1A2AE05FEC334:411790099DF40245691EEA02BF6069E5

The first part before the : is the hash, and second part is the salt.

When users log in, the hash is generated again each time, and instead of a random salt, the salt from the database is fetched, and the algorithm is used again. This is very clever, as no one except the customer will be able to know their password. An example of an early version was simply plaintext passwords, which is extremely unsafe.



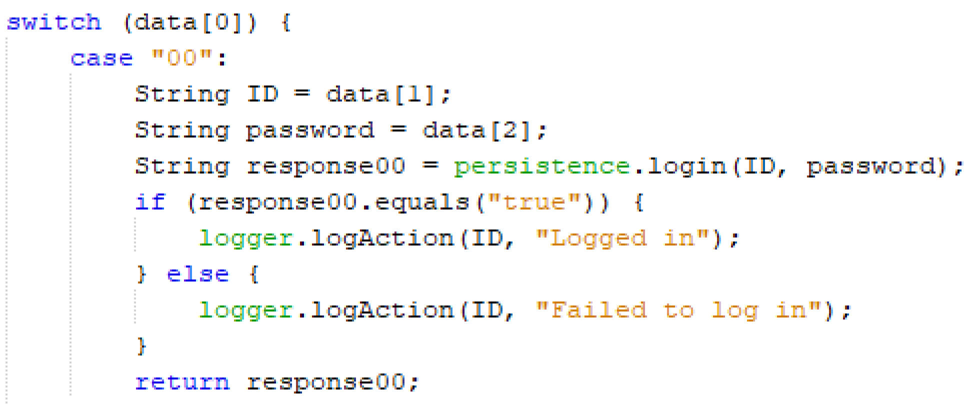
18	C1234957632	Rop Christensen	1985-03-09	64227892	AddresseSverige	Kim2@mail.dk	true	password1234

Once these methods had been made it was rather simple to call them again, whenever a password would need to be hashed, or validated.

8.5 Logging

Logging was implemented in order to see what the different users in the system does. By doing this, it is possible to keep track of what has happened and who did it. This is partially to comply with some of the GDPR rules[9] and also because it is useful to have a log in a banking system.

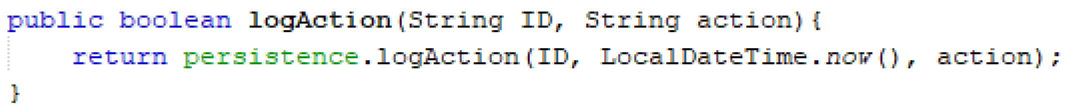
Logging is done by calling the logAction method in the ActionLogger class in the web-service code. This method takes two strings as parameters; the ID of the user and their action. The method then adds the date and sends it to the persistence layer, where an SQL query is made with the ID, date/time and an action. This is saved to the “logger” table in the database.
The calls to the ActionLogger is made in a java class, called ServerProtocol, in the web-service code that handles the incoming messages from the client. The message from the client complies with the “Bank of Elena protocol”. There is a switch-case that determines the action to take depending on the protocol.
The following code shows the case that handles login. The variable response00 will either be “true” or “false”, depending on if the login was successful or not. This is logged in the if-else statement with logger.logAction(). If the login was successful, the logger will say that the person with the ID logged in. If the login returned “false”, the logger will say that the person with the ID failed to log in.



```java
switch (data[0]) {
    case "00":
        String ID = data[1];
        String password = data[2];
        String response00 = persistence.login(ID, password);
        if (response00.equals("true")) {
            logger.logAction(ID, "Logged in");
        } else {
            logger.logAction(ID, "Failed to log in");
        }
        return response00;
```

[9] GDPR logging: https://logsentinel.com/gdpr-logging-requirements/

The logAction method takes the two parameters and adds the LocalDateTime.



```
public boolean logAction(String ID, String action){
    return persistence.logAction(ID, LocalDateTime.now(), action);
}
```

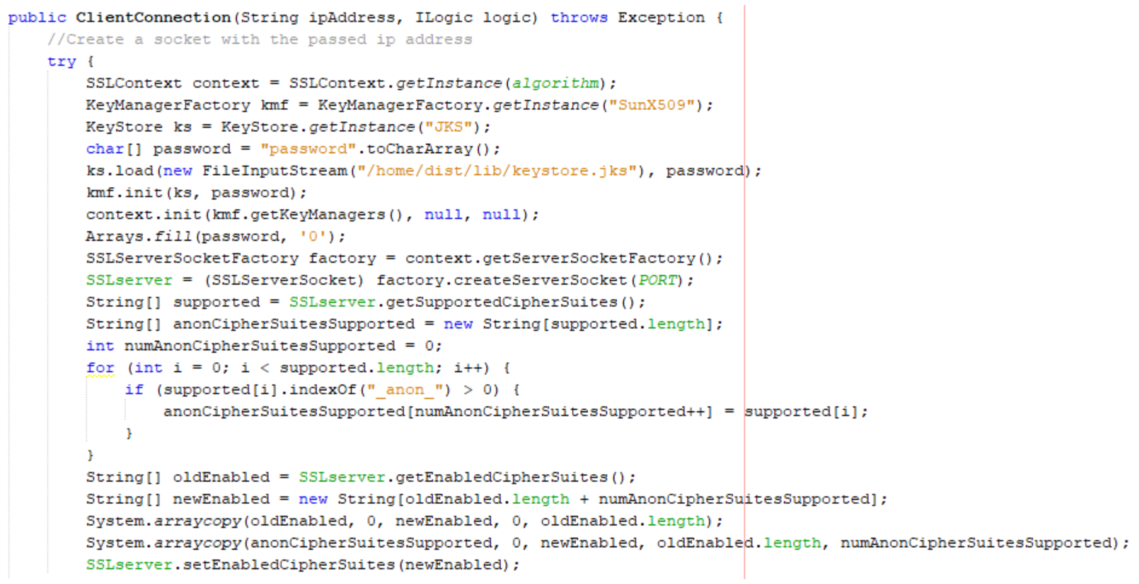
The DBmanager class in the persistence layer handles the connection to the database and it is here the SQL queries are written. A PreparedStatement with the INSERT query is made, with the variables from the logger, and executed on the database.

8.6 Secure Socket Layer / Transport Layer Security

As previously mentioned in the design part of the report, SSL/TLS was used as a way to ensure integrity and confidentiality, but how was this actually implemented, and did it end up working?

A major decision was to run the client in a java application instead of a browser. This decision had an impact on the way SSL/TLS had to be implemented as well. When using a browser, a webserver has to be running providing the frontend, which could for instance be NGINX or apache tomcat. In case of a signed certification, the browser accepts this by default without the end user having to accept it. With Java this is implemented in a more complex way, and once that has been done, the Java program has to use SSL sockets instead of the regular sockets which offer unencrypted data transfer.

With SSL sockets both the webservice and the client has to agree on how to establish connection in a more complicated way compared to normal sockets. The following code is part of the java implementation. The code is mostly from a book[10] but online research and previous experience by the project group ended up solving this task.



```
public ClientConnection(String ipAddress, ILogic logic) throws Exception {
    //Create a socket with the passed ip address
    try {
        SSLContext context = SSLContext.getInstance(algorithm);
        KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
        KeyStore ks = KeyStore.getInstance("JKS");
        char[] password = "password".toCharArray();
        ks.load(new FileInputStream("/home/dist/lib/keystore.jks"), password);
        kmf.init(ks, password);
        context.init(kmf.getKeyManagers(), null, null);
        Arrays.fill(password, '0');
        SSLServerSocketFactory factory = context.getServerSocketFactory();
        SSLserver = (SSLServerSocket) factory.createServerSocket(PORT);
        String[] supported = SSLserver.getSupportedCipherSuites();
        String[] anonCipherSuitesSupported = new String[supported.length];
        int numAnonCipherSuitesSupported = 0;
        for (int i = 0; i < supported.length; i++) {
            if (supported[i].indexOf("_anon_") > 0) {
                anonCipherSuitesSupported[numAnonCipherSuitesSupported++] = supported[i];
            }
        }
        String[] oldEnabled = SSLserver.getEnabledCipherSuites();
        String[] newEnabled = new String[oldEnabled.length + numAnonCipherSuitesSupported];
        System.arraycopy(oldEnabled, 0, newEnabled, 0, oldEnabled.length);
        System.arraycopy(anonCipherSuitesSupported, 0, newEnabled, oldEnabled.length, numAnonCipherSuitesSupported);
        SSLserver.setEnabledCipherSuites(newEnabled);
```

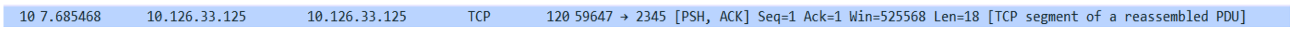
[10] Book source: Java Network Programming 4.th Edition - Eliotte Rusty Harold. P. 342

In short, what happens is that an encryption algorithm is chosen and then a way to manage the keys, which in this case is the SunX509 key manager factory[11].

Once that has been done a keystore, which is a list of private keys and corresponding certificates, is made which could be shared to clients for client authentication, opened and initialized in the SSL context. That context is then used to build an SSLServerSocket[12].
A list of supported ciphersuites is then run through to find different ways the client and server can agree on an encryption algorithm.
The client part is simpler, and code will not be shown here. A program called java keytool helped to build a new cacerts file which had the accepted certificate in it. The cacerts file was afterwards distributed to the client. It was a bit of a mixed solution as knowledge about it was limited, but in the end, it worked out.

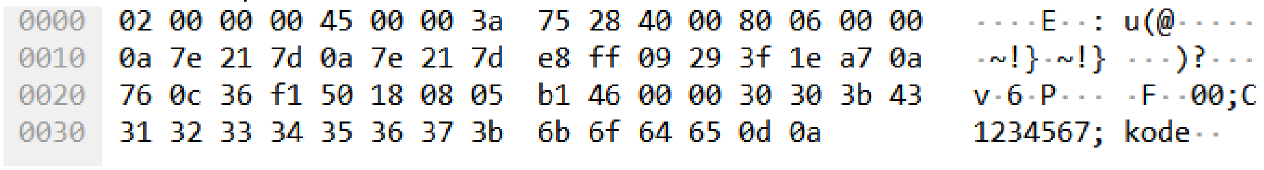
A security test was then performed before and after to compare the results, and to check whether the solution worked or not. First the program was running over normal sockets, and a program called Wireshark [13] was used to monitor network traffic.

Here is an example of the unencrypted data over TCP and how easy it was to sniff the password.



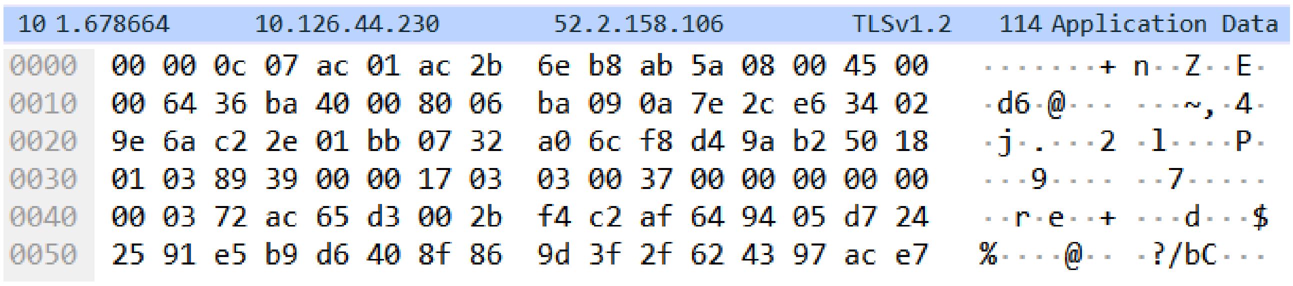
```
10 7.685468    10.126.33.125    10.126.33.125    TCP    120 59647 → 2345 [PSH, ACK] Seq=1 Ack=1 Win=525568 Len=18 [TCP segment of a reassembled PDU]
```

That is the packet which contained the login for the customer.
Here is the data it provided.



```
0000  02 00 00 00 45 00 00 3a  75 28 40 00 80 06 00 00   ····E··: u(@·····
0010  0a 7e 21 7d 0a 7e 21 7d  e8 ff 09 29 3f 1e a7 0a   ·~!}·~!} ···)?···
0020  76 0c 36 f1 50 18 08 05  b1 46 00 00 30 30 3b 43   v·6·P··· ·F··00;C
0030  31 32 33 34 35 36 37 3b  6b 6f 64 65 0d 0a         1234567; kode··
```

And here is an example of an encrypted data package and the data.



```
10 1.678664    10.126.44.230    52.2.158.106    TLSv1.2    114 Application Data

0000  00 00 0c 07 ac 01 ac 2b  6e b8 ab 5a 08 00 45 00   ·······+ n··Z··E·
0010  00 64 36 ba 40 00 80 06  ba 09 0a 7e 2c e6 34 02   ·d6·@··· ···~,·4·
0020  9e 6a c2 2e 01 bb 07 32  a0 6c f8 d4 9a b2 50 18   ·j·.···2 ·l····P·
0030  01 03 89 39 00 00 17 03  03 00 37 00 00 00 00 00   ···9···· ··7·····
0040  00 03 72 ac 65 d3 00 2b  f4 c2 af 64 94 05 d7 24   ··r·e··+ ···d···$
0050  25 91 e5 b9 d6 40 8f 86  9d 3f 2f 62 43 97 ac e7   %····@·· ·?/bC···
```

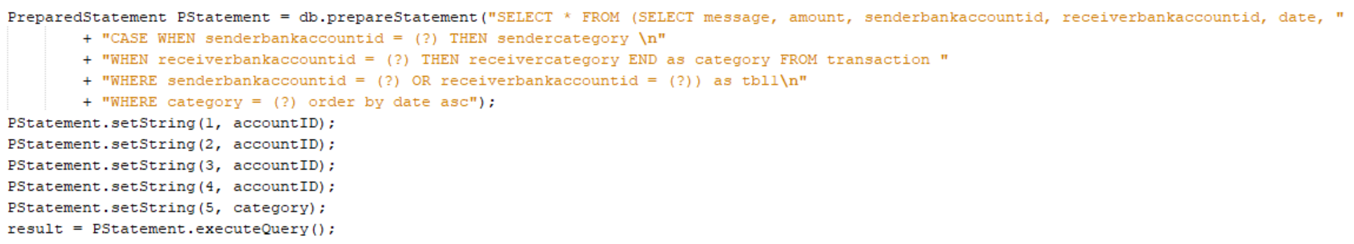
This clearly shows TLS has been established along with its version, which in this case is V1.2, and even if someone was snooping on the connection, they would not be able to read or decipher anything. The version number is rather important to notice, as security flaws have been found in for instance SSL 2 and SSL 3.

This concludes this section and leaves an important reminder to always use encrypted connections as public unencrypted networks will not protect against monitoring programs. This was recorded locally on a computer on an encrypted network called Eduroam. If normal sockets were used, that data would still be protected by eduroam, but it's a huge risk to allow banking info over normal sockets. For the full data capture, please check the files sent along with this report in the zip file.

[11] Further reading about SunX509:
https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html
[12]Java KeyStore: https://docs.oracle.com/javase/7/docs/api/java/security/KeyStore.html
[13] Network monitor: https://www.wireshark.org/

8.7 Transaction categories

Even though the data is secured by safe connections, encryption, hashing, and so on. There was a requirement regarding the possibility to make and change categories on transactions. This functionality was implemented towards the end of the project, but it was not an easy task. At the beginning, the project group didn't include a category column in the database scheme and therefore only a single transaction was shared between two users. This became a problem as it should now be possible to set and change a category. As stated earlier in this section, data confidentiality was an important focus, and with the system only having a single transaction, a category would be shared between two accounts. This was solved by making two columns, a sender and receiver category, and through a few complex sql queries, it was possible to only show the correct category to each user. They would thereby have no idea what the other involved person's category was. A total rework of the transaction history, and new implementations of change category, and set category made this possible.
This was an important reminder how much an early design choice impacts decisions later on, and while the project group strived to have the system easily modifiable, such a change still presented a tough challenge.
This code shows how the query differentiates between the sender and receiver category.



```
PreparedStatement PStatement = db.prepareStatement("SELECT * FROM (SELECT message, amount, senderbankaccountid, receiverbankaccountid, date, "
        + "CASE WHEN senderbankaccountid = (?) THEN sendercategory \n"
        + "WHEN receiverbankaccountid = (?) THEN receivercategory END as category FROM transaction "
        + "WHERE senderbankaccountid = (?) OR receiverbankaccountid = (?)) as tbl1\n"
        + "WHERE category = (?) order by date asc");
PStatement.setString(1, accountID);
PStatement.setString(2, accountID);
PStatement.setString(3, accountID);
PStatement.setString(4, accountID);
PStatement.setString(5, category);
result = PStatement.executeQuery();
```

For the full code, please refer to appendix F.

8.8 Docker

The Dockerfile which was used for building an image for the web-service can be seen in appendix B. The image is based on the prebuilt anapsix/alpine-java image. This image is a basic Docker image that includes java runtime environment and BASH. The next line in the Dockerfile is where our program is copied into the image. The dist folder contains the jar-file, the key for the socket encryption, and two Java libraries; javax.mail and postgresql. Then port 2345 is exposed as that is the port the java program listens on. Finally, when the container is run, the container executes the jar-file and the program starts and it is possible to connect through the client.

After the image was built, it was uploaded to Dockerhub, so it can easily be deployed anywhere. The image name is "peterzxcvbnm/webservice:finalv7". The container was then deployed to the server given to the group and port 2345 on the server was bound to the exposed port in the container.

8.9 Summary of implementation

To summarize implementation, the user interface was created by making an fxml file and a controller for every fxml file. Furthermore, the web-service and the client both follow the protocol. The protocol defines the rules for the communication, which made implementation efficient since people could work on different sections without having too many conflicts. The SQL statements to the database was made using PreparedStatements which helps to secure against SQL injection. To make the system even more secure, hashing was implemented using PBKDF2WithHmacSHA1, which generates a strong and secure hash value together with a salt. Information about what is happening in the system is essential for correcting errors which is why creating a log is beneficial. Furthermore, implementing SSL/TLS in java was done by using SSL sockets, an encryption algorithm, and a way to manage the keys, which is Sunx509. Furthermore, transaction categories were implemented through hard effort, but by reworking an SQL query, it was possible. Finally, the web-service was deployed using Docker.

9. Testing

This chapter will explain how the group handled testing to make sure that the system works as expected and how debugging was done when bugs/errors occurred.

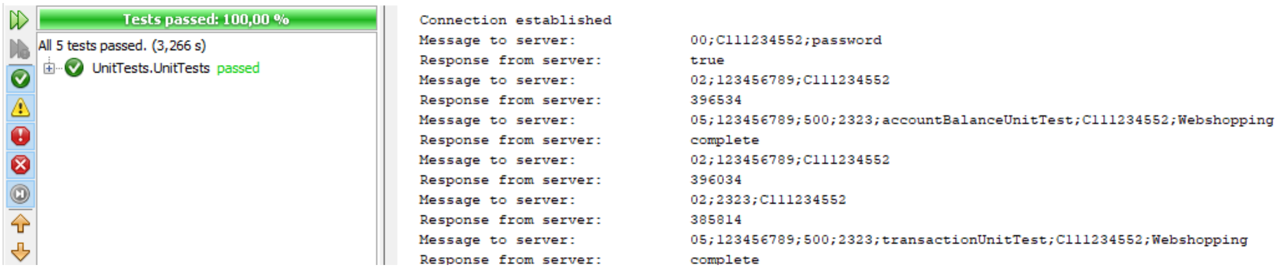
9.1 Debugging

There have been many different bugs during the programming of the banking system. These bugs needed to be fixed. To fix these bugs, the group used a “technique” where a lot of console printouts would be written between the different lines of code, to test where the bug happened.
If for example there was a problem with the connection, then there would be made many System.out.println statements between the actual code. This way the group could see what the last printout was and figure out which line of code broke the system.
Other times the group would print out some of the variables in the functionality that was being tested, to see whether it was the right variables or not. By looking at what was printed out in the console, the group could figure out why a lot of bugs happened.
There was also made use of try-catch statements to make the code more fail-safe. If the code would throw an exception because something went wrong, then the programmers could read the stacktrace or similar error handling methods to see what had happened and correct the error/bug.

Startup tests were implemented that checked most of the functionality when the program started, to make sure that the system always worked as intended. This way the programmers could see if the startup test passed or failed. To further be sure that there were no bugs, the program was also tested through the GUI to be completely sure that the program worked, because the startup test could not catch all bugs. Sometimes, for example, a test would pass but there would be a bug with the GUI.
The startup test only tested some of the methods in the code and therefore the group decided to make use of JUnit to make real unit-tests of the code, to further check for errors. The next section will go into more detail about these unit-tests.

9.2 Unit-tests

To make sure the system works correctly, some unit-tests have been made to test certain system functionalities. The functionalities in the system that have been tested are: transaction, login, logout, view account balance, and create bank account.
The test for transaction examines what the expected amount is after a transaction has been made compared to what the amount is after the transaction has been completed. If they are equal, the unit test prints out true, which means the unit test was a success. Another unit test that was made, is login, where the expected result is true. Then the login method is called, and the returned data is compared to the expected result. If they are equal, then the unit test was a success.
The same thing has been done for the three other unit tests, where there is an expected result that is being compared to the result that is returned when calling the method, and if the expected and actual result are equal, then the unit test is a success.



```
Connection established
Message to server:          00;C111234552;password
Response from server:       true
Message to server:          02;123456789;C111234552
Response from server:       396534
Message to server:          05;123456789;500;2323;accountBalanceUnitTest;C111234552;Webshopping
Response from server:       complete
Message to server:          02;123456789;C111234552
Response from server:       396034
Message to server:          02;2323;C111234552
Response from server:       385814
Message to server:          05;123456789;500;2323;transactionUnitTest;C111234552;Webshopping
Response from server:       complete
```

Figure 10 - Image showing the results of the unit-tests

9.3 Summary of testing

To summarize the testing section, there have been made five unit tests and a few start up tests. Debugging was done mainly through print outs and reading stack traces. The various bugs that popped up were dealt with, as they were discovered. The group has tried to make sure that the functionalities of the system work as expected without errors.

10. Market analysis

Nigeria is going to be the target of our potential expansion. Nigeria has been chosen, as Africa is developing economically at a rather quick pace. Nigeria is at the forefront of this development, so it seems an obvious choice as a first contact for entering the African market. First, Nigeria is going to be analysed using the CAGE distance framework[14] to lay a groundwork for the rest of the analysis. Then, it is going to be followed up by an analysis of buyer behaviour to establish the expected reaction from potential customers. Finally, a risk assessment is going to be created to complete the analysis.

10.1 CAGE distance framework

The CAGE distance framework is a framework that looks at the differences within the four aspects; culture, administration, geographic location, and economics.

[14] CAGE framework: https://www.ghemawat.com/wordpress/wp-content/uploads/2016/10/DifferencesAndTheCAGEFramework.pdf

Cultural distance

One of the cultural aspects with the greatest influence is language. While Nigeria has upwards of 500 different languages, they still have English as their official language and many speak it as a second language. This means there is a language gap, but it could be worse. It is also worth noting that people might be better at English in urban areas compared to rural areas, so it might be a good idea to focus on urban customers first.
While there, of course, are many other cultural differences, they do not have direct impact on the system as it will have minimal face to face customer interaction and this could potentially be the responsibility of a local office.

Administrative distance

The EU and ECOWAS, which is the trading bloc that Nigeria is a member of, have good relations. Furthermore, the EU is the biggest importer of Nigerian goods[15]. This is also the reason an economic partnership agreement is being worked on. If this is agreed upon, it would further shorten the administrative distance.
The major administrative obstacle is corruption. While Nigeria has strong anti-corruption laws, they are weakly enforced, and bribery is the norm[16]. This is very different from Denmark and thus might be one of the most difficult things to get accustomed to.

Geographic distance

While Nigeria is a much larger country both in size and population than Denmark, it is not very far away and still in the same time zone. This means it should be possible to respond to problems with the system in a timely manner, as the problems are most likely to occur within normal business hours. The problem of handling more customers has been examined previously in the scalability section.

Economic distance

Denmark has a 9 times higher gross domestic product per capita than Nigeria when corrected for purchasing power. This paired with a higher income inequality means that the economic landscape is vastly different[17]. The Bank of Elena should therefore be expecting higher variety in the expectations of their customers depending on their wealth, which might lead to more requirements regarding customization.

10.2 Buyer behaviour

Nigerians prefer to keep money at home instead of in banks. This is due to a few different reasons. Some believe that it is due to corruption and that Nigerians believe the finance sector to be greedy. Financial analyst Rotimi Agboluaje identifies “illiteracy, lack of financial education and lack of confidence in the banking sector as reasons many Nigerians keep money at home”[18]. It obviously talks against entering the market. However, being aware of it, it might be used as an advantage.

[15] Nigeria and the EU: https://eeas.europa.eu/generic-warning-system-taxonomy/404/1621/nigeria-and-eu_en
[16] Corruption in Nigeria: https://www.business-anti-corruption.com/country-profiles/nigeria/
[17] Economic statistics: https://tradingeconomics.com/
[18] Nigerian banking: https://www.premiumtimesng.com/business/business-data/214749-nigerians-now-prefer-keep-money-home-instead-banks.html

If trust can be established with customers, it would be a huge competitive advantage over other banks.
Furthermore, many Nigerians only have access to the internet through their phones and thus the majority of their banking will be done through a phone. This is an issue, as the client was developed for computers. A mobile app would be better for Nigerian customers.

10.3 Risk assessment

There are a number of risks associated with entering Nigeria. Looking at both Hofstede's and the GLOBE dimensions, the biggest gap is power distance. The next two dimensions with differences are individualism/collectivism and uncertainty avoidance. These are all crucial to be aware of, as they are also the aspects that have the greatest impact on company cooperation and internal culture. These dimensions exemplify some of the local norms and values and breaking these could cause trust issues and conflicts among employees.

10.4 Analysis conclusion

Nigeria is a very risky market to enter. They have a very different economic and administrative environment compared to Denmark. This coupled with very different social norms and expectations poses many challenges. The greatest challenge for the system would probably be the general inclination of the public to keep money at home instead of in banks. However, Nigeria is also a country in fast development with many initiatives to ease the possibility of foreign investment.
To successfully deploy our software in Nigeria, we would need to have the processes behind our software be transparent in an effort to try to establish trust with potential customers.
Additionally, it is very important to develop a mobile client for the potential customers were a mobile phone is their only internet access.

11. Reflection

This chapter will focus on the group's reflections about the system, how to manage a project and software development.

11.1 System shortcomings

As with all things, the system has some flaws and shortcomings. The ones mentioned here mainly exist due to the short development time and have therefore been deprioritized to make room for more critical features, both in terms of functionality, but also to better display the skill of the group.

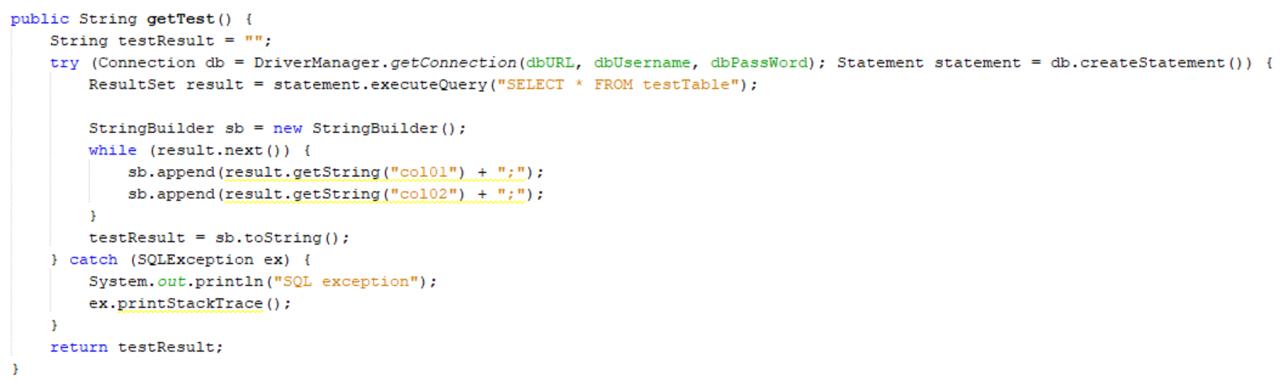
As the group consists of seven different people, there is bound to be some code inconsistency. Method and variable names might not always be similar, even though they all follow the java conventions. All the members have different habits in terms of commenting and thus some code contains more comments than other code.

The system also has a minor problem concerning admin accounts. At the moment, there is only one accessible admin account. The only way to add another is to call the hashing method and have it print out a hashed password and then manually add the account using SQL. This is obviously not ideal, but the group chose to focus on the customer experience and making the system secure instead.

11.2 Organizing & cleaning the code

As the system grew and more functionalities got added and changed, parts of the code became unused and outdated. However, those parts were not removed. This consequently led to unused and outdated methods, parameters, and objects. These could be cleaned up to make the code easier to read. Another thing is that the code itself is not very organized. When writing the code, members of the group just created methods or initialized objects wherever they felt like they belonged. The group did try to clean up the messageparser class in the client, so that it resembles how the protocol is structured.

An example of an old test method used to retrieve data from the database can be seen in the code below. At the time of implementation, one could copy the method and then change the name and the query. Instead of rewriting it all again, there was just one easy method to copy. However later in the project, the group switched to using prepared statements instead, so this method became obsolete. It also never actually had any real functionality for the system to work, so it is only taking up space right now.



```
public String getTest() {
    String testResult = "";
    try (Connection db = DriverManager.getConnection(dbURL, dbUsername, dbPassWord); Statement statement = db.createStatement()) {
        ResultSet result = statement.executeQuery("SELECT * FROM testTable");

        StringBuilder sb = new StringBuilder();
        while (result.next()) {
            sb.append(result.getString("col01") + ";");
            sb.append(result.getString("col02") + ";");
        }
        testResult = sb.toString();
    } catch (SQLException ex) {
        System.out.println("SQL exception");
        ex.printStackTrace();
    }
    return testResult;
}
```

11.3 Continued development

As with all things, the Bank of Elena system could be improved in a number of different ways. Apart from the functional requirements, which was not implemented due to time constraints, there are a number of other improvements that could be made to the system. Many of these were not implemented, because the group chose to focus on other aspects of the system. Others could simply not be completed in time. That said, it is still worth considering the possibilities if the system were to be deployed in a commercial setting.

Load balancer

Almost every online system sits behind a load balancer these days. Load balancers offer several benefits to a system. Its main task is, of course, to distribute client requests to several servers. This has the added benefit that if one server malfunctions, other servers are immediately ready to take over any requests without the need for human intervention. Furthermore, advanced load balancers can be a crucial security measure against DDOS attacks.
A load balancer was not implemented as the group felt it was out of scope of the project.

Database synchronisation

The only synchronization of the system is done in the code of the web-service. As previously mentioned, this could become a problem if the system was to be scaled up with several instances of the web-service running at the same time. To prevent this causing issues, one could change the database to be event controlled. This means that instead of having individual queries, all database requests would be added to a transaction log. These events are independent of each other, so it does not matter which order they are processed in.

Let us use a simple money transfer as an example. In our system, the web-service first reads the balance of both the receiving and sending account. Then the new balances are calculated. Finally, the database is updated with the new balances. This leaves plenty of time for the account balance to have been changed in the meantime if another web-service was updating the bank accounts at the same time.

In an event based database, this would all be considered one event. So instead of the code calculating the new balances, the database would simply subtract/add the transfer amount to each bank account at the same time. This has the added benefit that it does not matter when the transaction occurs, as the event is independent of the amount of money in the account.

Database encryption

Right now, everything in the database, except passwords, has been saved in clear text that means that if anyone got access to the database, they would be able to see and retrieve every single piece of information that is being kept about our customers and admins. As the system do save some sensitive information about our customers and admins, this is not the best possible storage solution. So, one thing that could have been done to safeguard the information is to encrypt all the sensitive data in the database, and only retrieve and decrypt it, when it is needed. Also keeping the encryption/decryption method secret, so if a hacker gets access to the database, all they would get is encrypted data, which they cannot use. Doing this would make it so the web-service has to encrypt and decrypt all communication to and from the database. This makes the communication slower, but it’s a good trade-off for a safer database.

12. Conclusion

The banking system has been implemented as a three-tiered system with a client, web-service, and database as the three tiers. This was done to meet the requirements of the assignment, but also due to it improving the security of the database, because it is only the web-service which accesses the database and thus the developers have more control over user communication with the database. A protocol has been created to facilitate communication between the client and the web-service. The protocol is rather simple, which makes it easier to expand upon.

Both the client and the web-service follow a 3-layer architecture to improve readability, modifiability, and maintainability. Multithreading has been implemented to enable more users to connect to the web-service at the same time. However, this created potential concurrency issues if multiple users tried to send money at the same time. The transfer method is limited to one thread at a time to combat this issue using the synchronized keyword in java.

The GUI for the client was developed in fxml using JavaFX Scene Builder. Scene Builder makes GUI design quick and easy without having to manually write all the code. The GUI was developed to be intuitive to use, but it still has a focus on security with all inputs being sanitised and as little personal information being displayed as possible.

Even though user inputs are sanitised, interaction with the database is still done using prepared statements to prevent SQL injection. The database also keeps a log of actions done by the web-service and who made the requests.

The major security feature of the system is the use of SSL/TLS encryption for communication between the client and the web-service. This is implemented using java’s SSL socket class.

The web-service was then deployed by customising a preconstructed Docker image. A custom image was built by copying the jar file and the necessary additional files into the preconstructed image. The custom image was then used to create a container on the server made available for the project by SDU.

Manual testing was done throughout the development and any bugs discovered would mostly be debugged as soon as possible. Debugging was done both through strategic use of print outs and reading stack traces. A few unit tests were also implemented to test core functionality.

Finally, a market analysis was conducted to determine whether the system could be further developed and then deployed in another country. The country chosen for the analysis was Nigeria. The analysis concluded that Nigeria has a lot of economic potential and plenty of customers, but that there are several cultural differences that might become major challenges if the system were to be deployed in Nigeria.

The group used Scrum to manage the working process and to ensure tasks were completed. Trello was used as a tool to manage both the product and sprint backlogs. Furthermore, Trello enabled group members to show what they were currently working on to prevent two members working on the same thing. Each sprint was planned to roughly follow the phases of Unified Process.

13. References

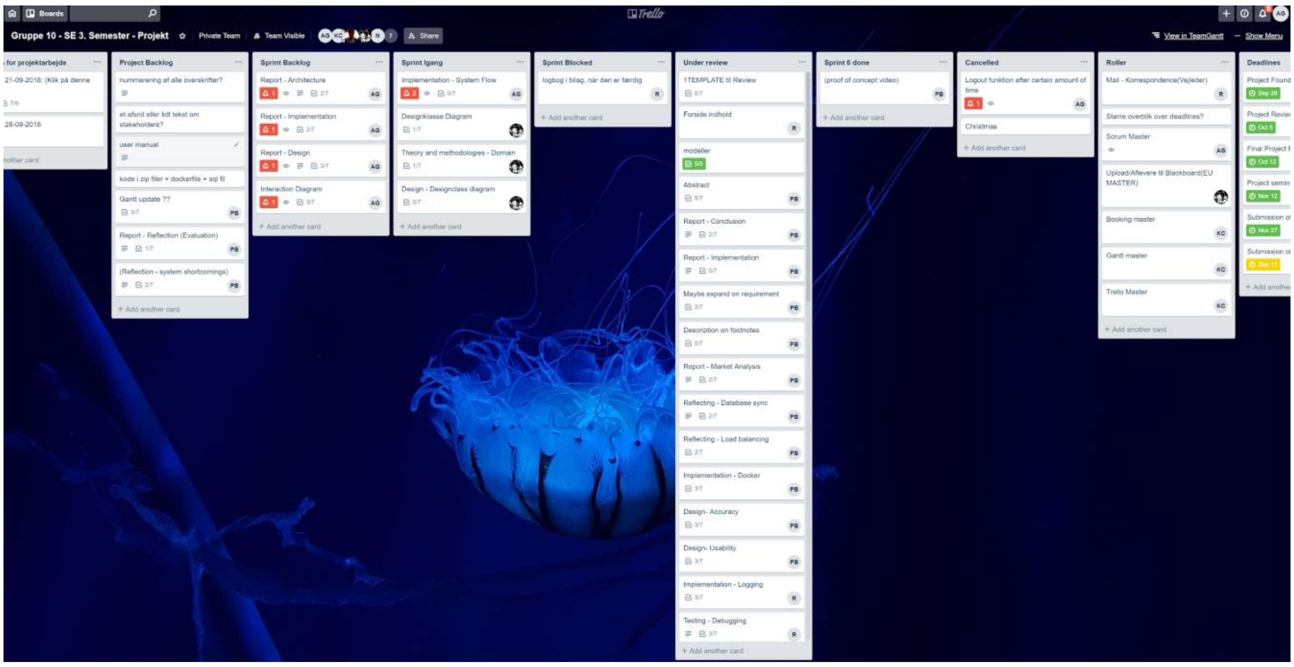
Websites

- https://www.ghemawat.com/wordpress/wp-content/uploads/2016/10/DifferencesAndTheCAGEFramework.pdf
- https://www.state.gov/e/eb/rls/othr/ics/2018/af/281450.htm
- https://eeas.europa.eu/generic-warning-system-taxonomy/404/1621/nigeria-and-eu_en
- https://tradingeconomics.com
- https://www.premiumtimesng.com/business/business-data/214749-nigerians-now-prefer-keep-money-home-instead-banks.html
- https://www.hofstede-insights.com/product/compare-countries/
- https://globeproject.com/results/countries
- https://www.worldfinance.com/banking/leading-nigerian-banking-into-the-digital-age
- https://hub.docker.com/r/anapsix/alpine-java/

14. Appendices

Appendix A - Trello

Overview of trello



Gruppe 10 - SE 3. Semester - Projekt

for projektarbejde
- 21-09-2018: (Klik på denne — 7/9
- 28-09-2018

Project Backlog
- nummerering af alle overskrifter?
- et afsnit eller lidt tekst om stakeholders?
- user manual
- kode i zip filer + dockerfile + sql fil
- Gantt update ?? — 3/7 — PB
- Report - Reflection (Evaluation) — 1/7 — PB
- (Reflection - system shortcomings) — 2/7 — PB

Sprint Backlog
- Report - Architecture — 1 — 2/7 — AG
- Report - Implementation — 1 — 2/7 — AG
- Report - Design — 1 — 3/7 — AG
- Interaction Diagram — 1 — 3/7 — AG

Sprint Igang
- Implementation - System Flow — 2 — 0/7 — AG
- Designklasse Diagram — 1/7
- Theory and methodologies - Domain — 1/7
- Design - Designclass diagram — 0/7

Sprint Blocked
- logbog i bilag, når den er færdig — R

Under review
- 1TEMPLATE til Review — 0/7
- Forside indhold — R
- modeller — 5/5
- Abstract — 5/7 — PB
- Report - Conclusion — 2/7 — PB
- Report - Implementation — 0/7 — PB
- Maybe expand on requirement — 3/7 — PB
- Description on footnotes — 0/7 — PB
- Report - Market Analysis — 2/7 — PB
- Reflecting - Database sync — 2/7 — PB
- Reflecting - Load balancing — 2/7 — PB
- Implementation - Docker — 3/7 — PB
- Design- Accuracy — 3/7 — PB
- Design- Usability — 3/7 — PB
- Implementation - Logging — 3/7 — R
- Testing - Debugging — 3/7 — R

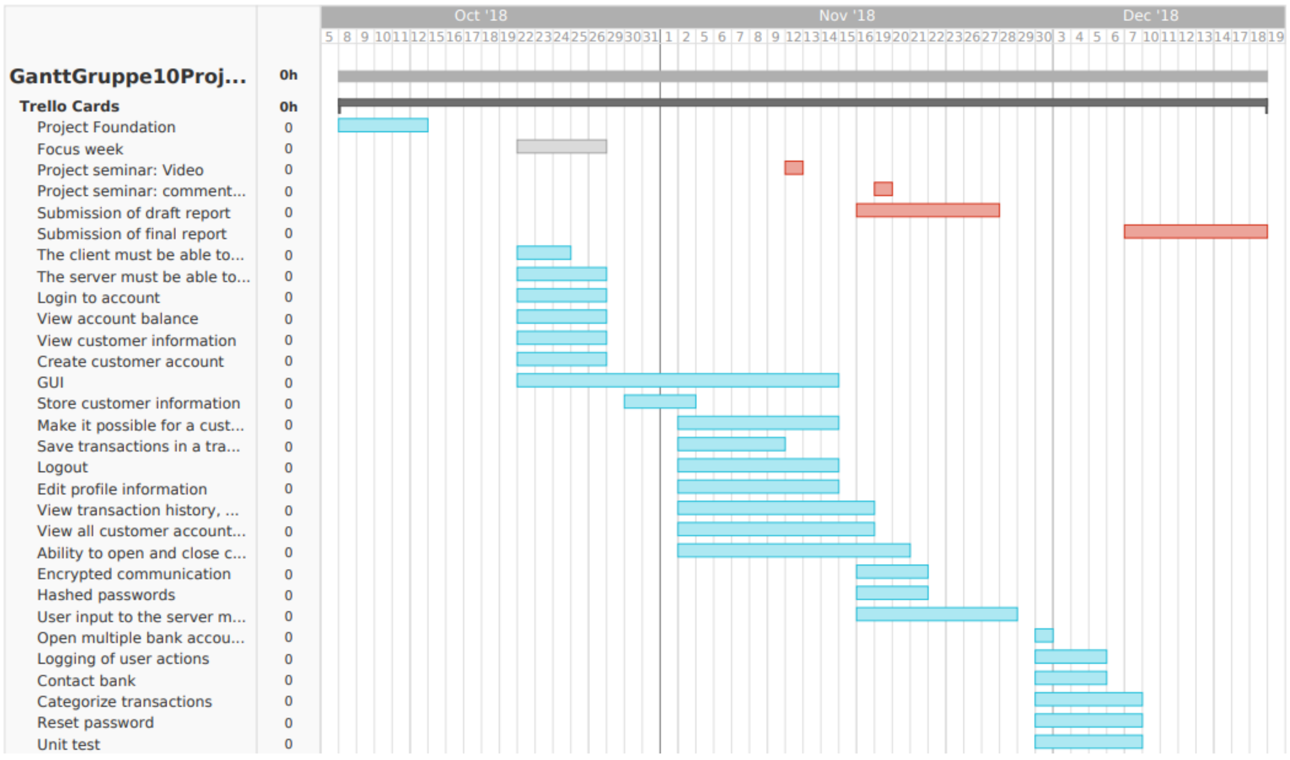
Sprint 6 done
- (proof of concept video) — PB

Cancelled
- Logout funktion after certain amount of time — 1 — AG
- Christmas

Roller
- Mail - Korrespondence(Vejleder) — R
- Større overblik over deadlines?
- Scrum Master — AG
- Upload/Aflevere til Blackboard(EU MASTER)
- Booking master — KC
- Gantt master — KC
- Trello Master — KC

Deadlines
- Project Found — Sep 28
- Project Revie — Oct 5
- Final Project F — Oct 12
- Project semin — Nov 12
- Submission of — Nov 27
- Submission of — Dec 17

Gantt diagram



GanttGruppe10Proj...	0h
Trello Cards	0h
Project Foundation	0
Focus week	0
Project seminar: Video	0
Project seminar: comment...	0
Submission of draft report	0
Submission of final report	0
The client must be able to...	0
The server must be able to...	0
Login to account	0
View account balance	0
View customer information	0
Create customer account	0
GUI	0
Store customer information	0
Make it possible for a cust...	0
Save transactions in a tra...	0
Logout	0
Edit profile information	0
View transaction history, ...	0
View all customer account...	0
Ability to open and close c...	0
Encrypted communication	0
Hashed passwords	0
User input to the server m...	0
Open multiple bank accou...	0
Logging of user actions	0
Contact bank	0
Categorize transactions	0
Reset password	0
Unit test	0

Appendix B - Dockerfile

```
FROM anapsix/alpine-java

COPY dist /home/dist

EXPOSE 2345

CMD ["java","-jar","/home/dist/GLO18_WebService.jar"]
```

Appendix C - Detailed use case descriptions

View Transaction History

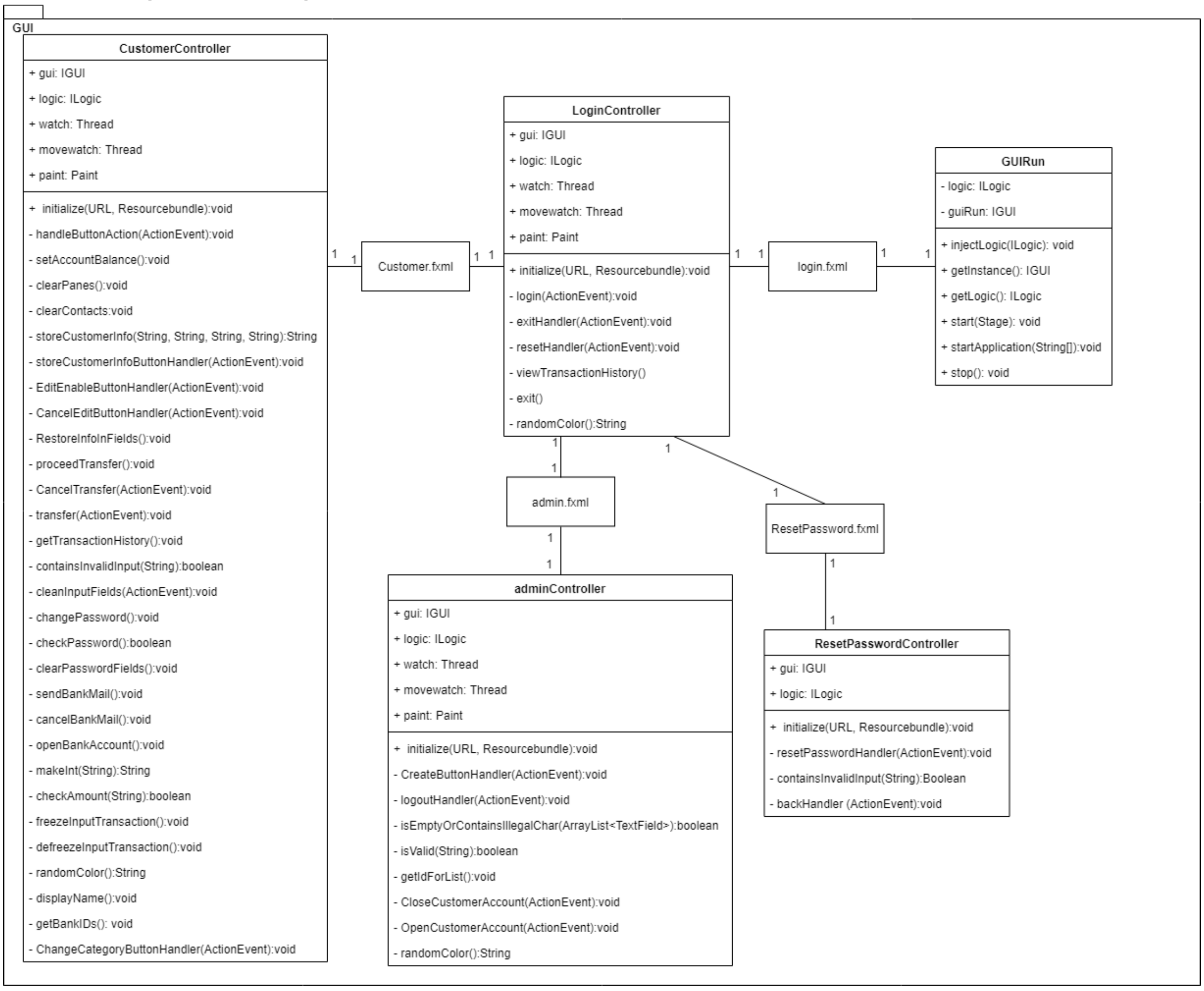
Use case: View Transaction History
ID: 3
Brief description: The customer views the transaction history
Primary actors: Customer
Secondary actors: None
Preconditions: The customer is logged in. The customer has a bank account.
Main flow: 1. The use case starts when the customer wants to view the transaction history 2. The client makes a request to the web-service to view the transaction history for the customer 3. The webservice makes a request to the database and gets the transaction history for the customer 4. The web-service sends the transaction history to the client 5. The client displays the transaction history
Postconditions: The transaction history is displayed
Alternative flows: None

Create customer account

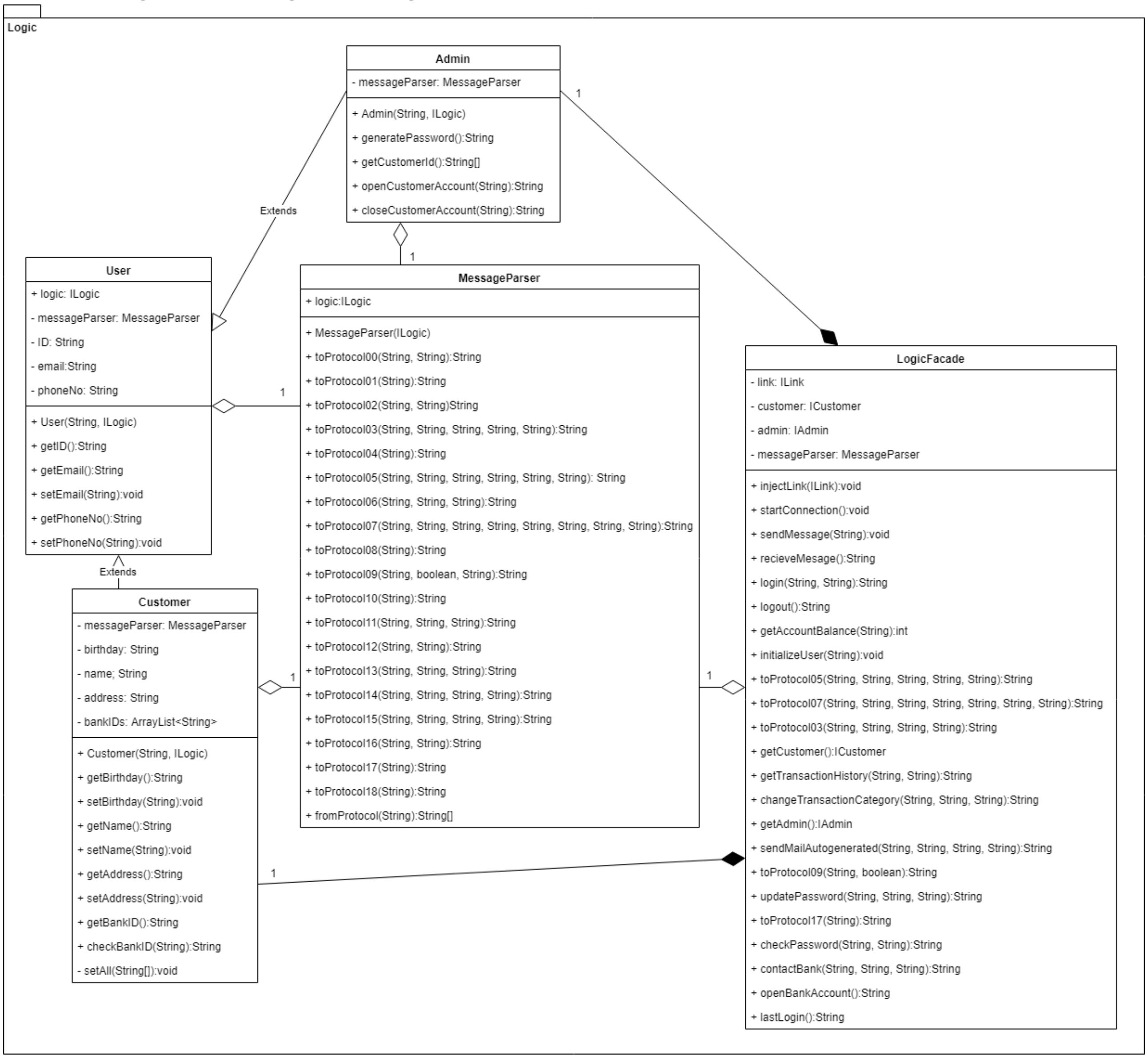
Use case: Create customer account
ID: 4
Brief Description: An administrator creates a customer account
Primary actors: Administrator
Secondary actors: None
Preconditions: The administrator must have the information for the customer account. The Admin must be logged into the system.
Main flow: 1. The use case starts when the administrator needs to create a new customer account 2. If administrator fills in relevant information and presses submit   2.1. If the entered information matches expected input   2.1.1. The client makes a server request   2.1.2. If the request is successful   2.1.2.1. The administrator is notified that the user was created and can be seen by the admin   2.1.2.2. The customer is automatically sent an email with the password   2.1.3. Else an error is displayed regarding the failed request attempt   2.2. Else an error is displayed regarding wrong input. 3. Else an error is displayed regarding missing input
Postconditions: A customer account is created and can be seen by the admin. The customer automatically received an email with login information.
Alternative flows: None

Appendix D - Detailed design class diagrams

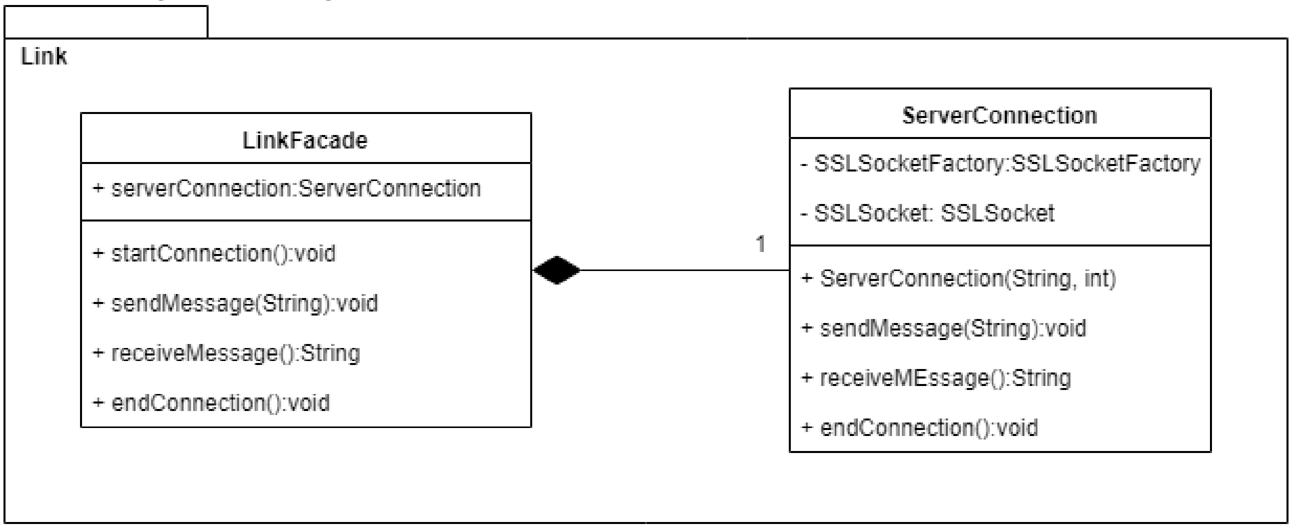
Client design class diagram - GUI


GUI
CustomerController
+ gui: IGUI
+ logic: ILogic
+ watch: Thread
+ movewatch: Thread
+ paint: Paint
+ initialize(URL, Resourcebundle):void
- handleButtonAction(ActionEvent):void
- setAccountBalance():void
- clearPanes():void
- clearContacts:void
- storeCustomerInfo(String, String, String, String):String
- storeCustomerInfoButtonHandler(ActionEvent):void
- EditEnableButtonHandler(ActionEvent):void
- CancelEditButtonHandler(ActionEvent):void
- RestoreInfoInFields():void
- proceedTransfer():void
- CancelTransfer(ActionEvent):void
- transfer(ActionEvent):void
- getTransactionHistory():void
- containsInvalidInput(String):boolean
- cleanInputFields(ActionEvent):void
- changePassword():void
- checkPassword():boolean
- clearPasswordFields():void
- sendBankMail():void
- cancelBankMail():void
- openBankAccount():void
- makeInt(String):String
- checkAmount(String):boolean
- freezeInputTransaction():void
- defreezeInputTransaction():void
- randomColor():String
- displayName():void
- getBankIDs(): void
- ChangeCategoryButtonHandler(ActionEvent):void
Customer.fxml
LoginController
+ gui: IGUI
+ logic: ILogic
+ watch: Thread
+ movewatch: Thread
+ paint: Paint
+ initialize(URL, Resourcebundle):void
- login(ActionEvent):void
- exitHandler(ActionEvent):void
- resetHandler(ActionEvent):void
- viewTransactionHistory()
- exit()
- randomColor():String
login.fxml
GUIRun
- logic: ILogic
- guiRun: IGUI
+ injectLogic(ILogic): void
+ getInstance(): IGUI
+ getLogic(): ILogic
+ start(Stage): void
+ startApplication(String[]):void
+ stop(): void
admin.fxml
ResetPassword.fxml
adminController
+ gui: IGUI
+ logic: ILogic
+ watch: Thread
+ movewatch: Thread
+ paint: Paint
+ initialize(URL, Resourcebundle):void
- CreateButtonHandler(ActionEvent):void
- logoutHandler(ActionEvent):void
- isEmptyOrContainsIllegalChar(ArrayList<TextField>):boolean
- isValid(String):boolean
- getIdForList():void
- CloseCustomerAccount(ActionEvent):void
- OpenCustomerAccount(ActionEvent):void
- randomColor():String
ResetPasswordController
+ gui: IGUI
+ logic: ILogic
+ initialize(URL, Resourcebundle):void
- resetPasswordHandler(ActionEvent):void
- containsInvalidInput(String):Boolean
- backHandler (ActionEvent):void
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1

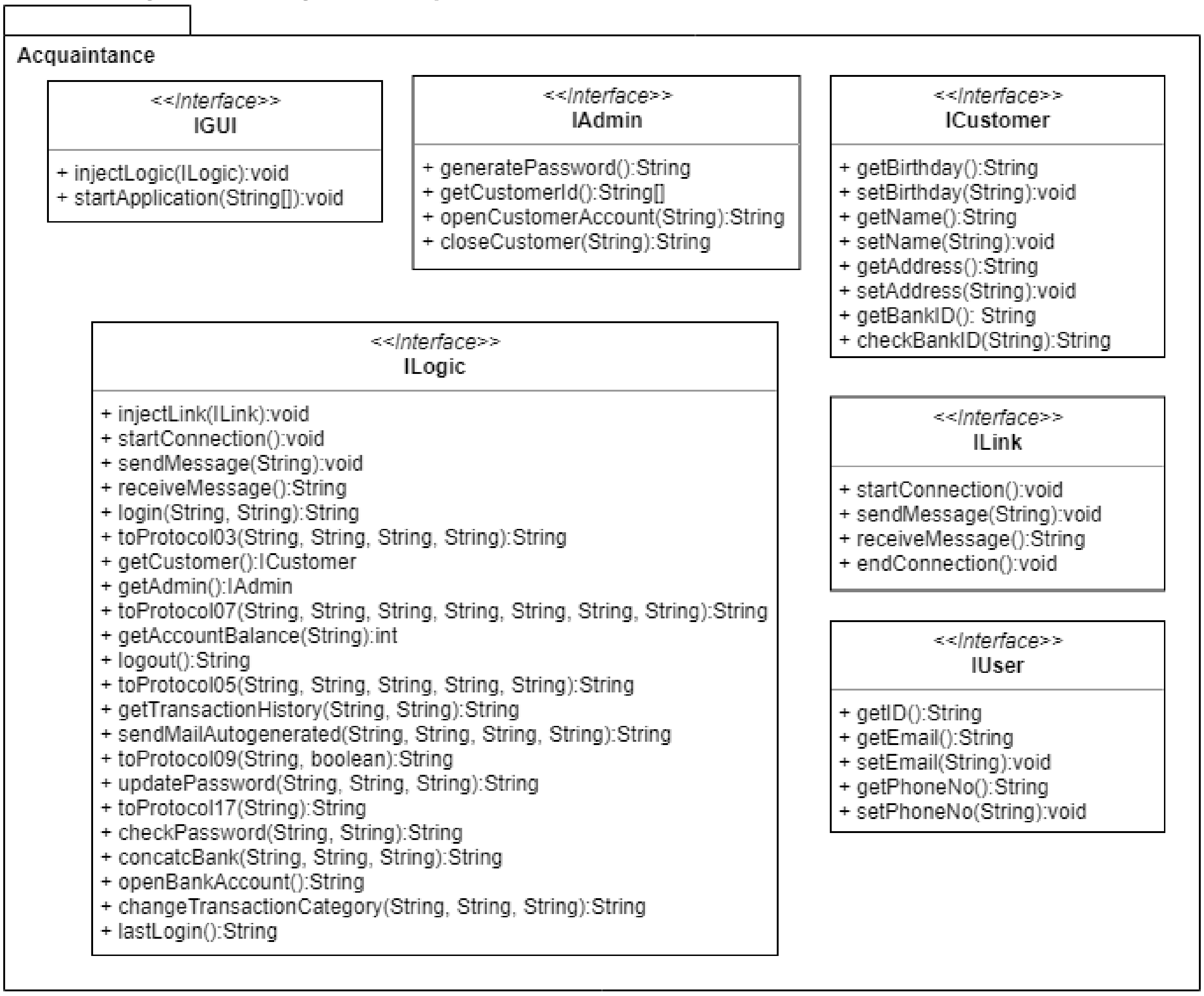
Client design class diagram - Logic


Logic
Admin
- messageParser: MessageParser
+ Admin(String, ILogic)
+ generatePassword():String
+ getCustomerId():String[]
+ openCustomerAccount(String):String
+ closeCustomerAccount(String):String
Extends
1
1
User
+ logic: ILogic
- messageParser: MessageParser
- ID: String
- email:String
- phoneNo: String
+ User(String, ILogic)
+ getID():String
+ getEmail():String
+ setEmail(String):void
+ getPhoneNo():String
+ setPhoneNo(String):void
Extends
MessageParser
+ logic:ILogic
+ MessageParser(ILogic)
+ toProtocol00(String, String):String
+ toProtocol01(String):String
+ toProtocol02(String, String)String
+ toProtocol03(String, String, String, String, String):String
+ toProtocol04(String):String
+ toProtocol05(String, String, String, String, String, String): String
+ toProtocol06(String, String, String):String
+ toProtocol07(String, String, String, String, String, String, String, String):String
+ toProtocol08(String):String
+ toProtocol09(String, boolean, String):String
+ toProtocol10(String):String
+ toProtocol11(String, String, String):String
+ toProtocol12(String, String):String
+ toProtocol13(String, String, String):String
+ toProtocol14(String, String, String, String):String
+ toProtocol15(String, String, String, String):String
+ toProtocol16(String, String):String
+ toProtocol17(String):String
+ toProtocol18(String):String
+ fromProtocol(String):String[]
1
1
1
Customer
- messageParser: MessageParser
- birthday: String
- name; String
- address: String
- bankIDs: ArrayList<String>
+ Customer(String, ILogic)
+ getBirthday():String
+ setBirthday(String):void
+ getName():String
+ setName(String):void
+ getAddress():String
+ setAddress(String):void
+ getBankID():String
+ checkBankID(String):String
- setAll(String[]):void
1
LogicFacade
- link: ILink
- customer: ICustomer
- admin: IAdmin
- messageParser: MessageParser
+ injectLink(ILink):void
+ startConnection():void
+ sendMessage(String):void
+ recieveMesage():String
+ login(String, String):String
+ logout():String
+ getAccountBalance(String):int
+ initializeUser(String):void
+ toProtocol05(String, String, String, String, String):String
+ toProtocol07(String, String, String, String, String, String, String):String
+ toProtocol03(String, String, String, String):String
+ getCustomer():ICustomer
+ getTransactionHistory(String, String):String
+ changeTransactionCategory(String, String, String):String
+ getAdmin():IAdmin
+ sendMailAutogenerated(String, String, String, String):String
+ toProtocol09(String, boolean):String
+ updatePassword(String, String, String):String
+ toProtocol17(String):String
+ checkPassword(String, String):String
+ contactBank(String, String, String):String
+ openBankAccount():String
+ lastLogin():String

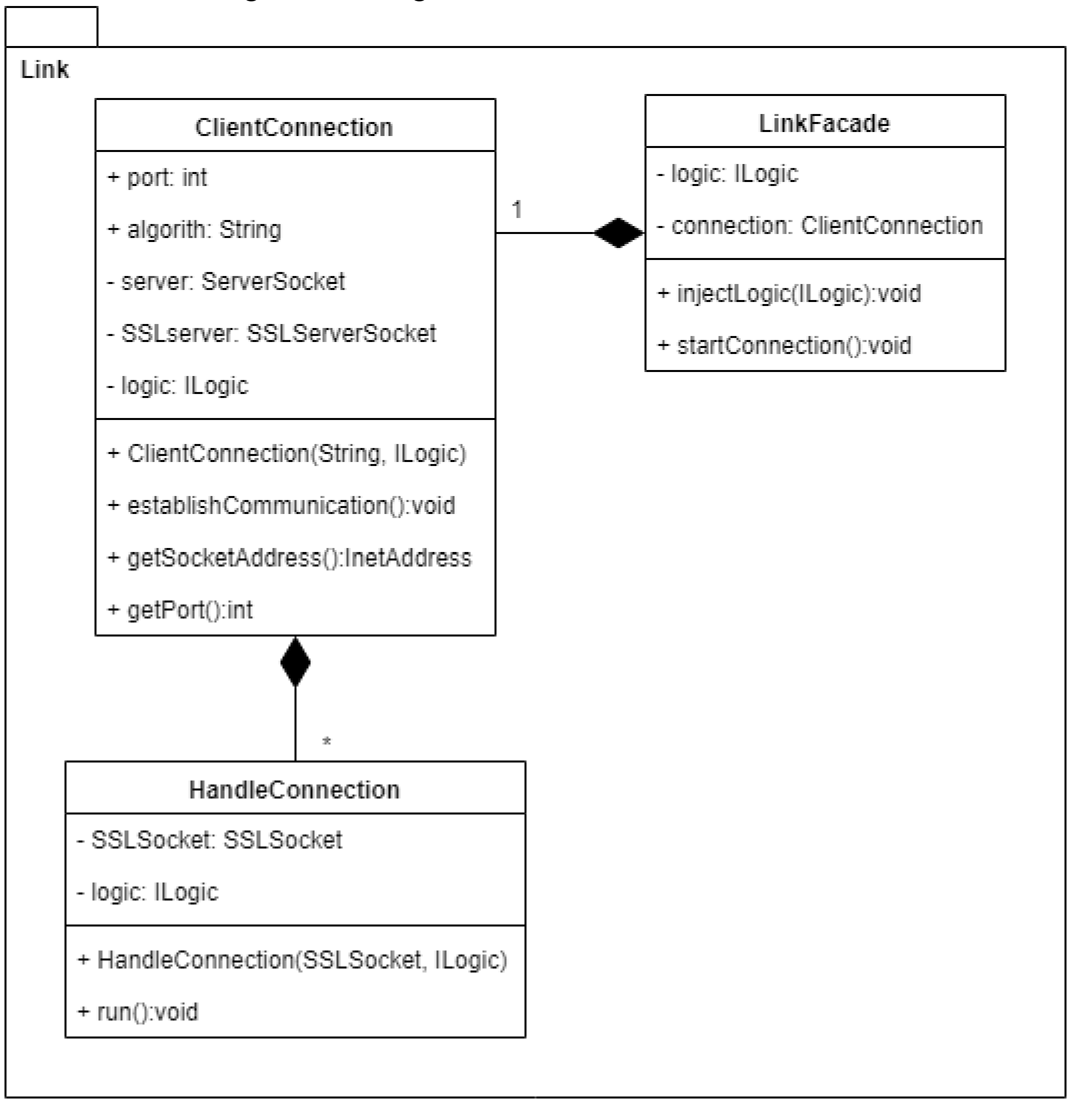
Client design class diagram - Link


Link
LinkFacade
+ serverConnection:ServerConnection
+ startConnection():void
+ sendMessage(String):void
+ receiveMessage():String
+ endConnection():void
1
ServerConnection
- SSLSocketFactory:SSLSocketFactory
- SSLSocket: SSLSocket
+ ServerConnection(String, int)
+ sendMessage(String):void
+ receiveMEssage():String
+ endConnection():void

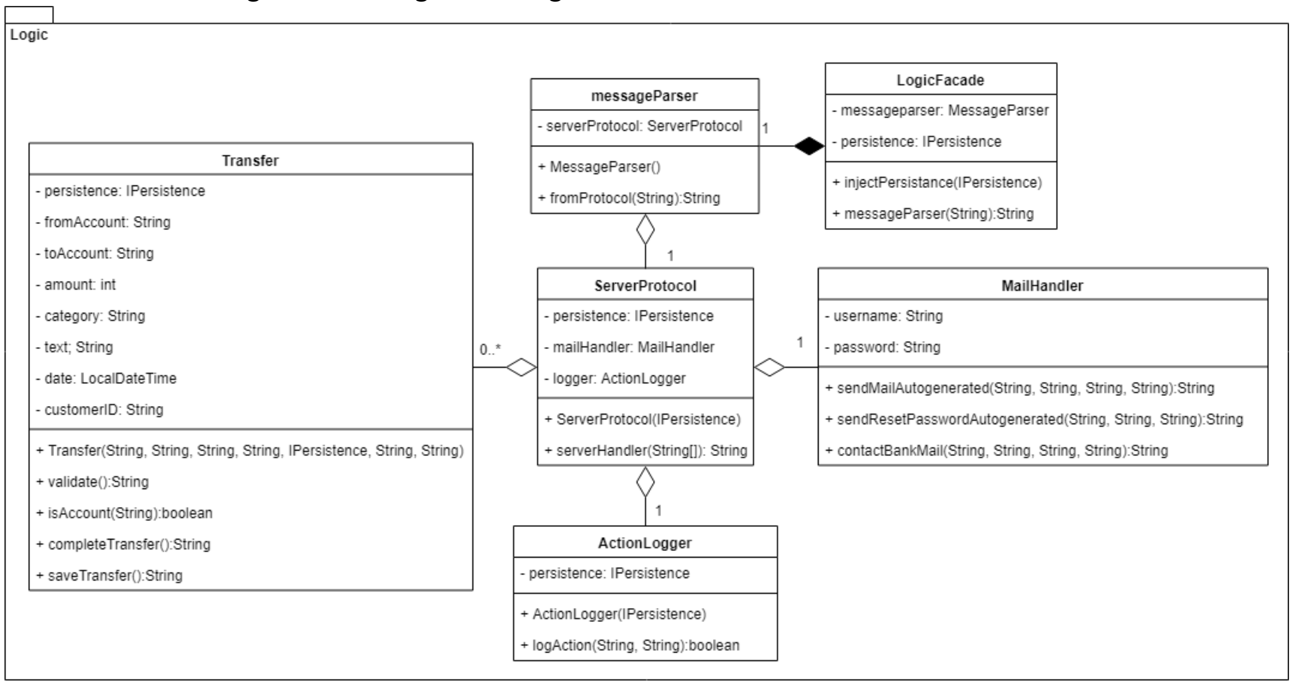
Client design class diagram - Acquaintance


Acquaintance
<<Interface>>
IGUI
+ injectLogic(ILogic):void
+ startApplication(String[]):void
<<Interface>>
IAdmin
+ generatePassword():String
+ getCustomerId():String[]
+ openCustomerAccount(String):String
+ closeCustomer(String):String
<<Interface>>
ICustomer
+ getBirthday():String
+ setBirthday(String):void
+ getName():String
+ setName(String):void
+ getAddress():String
+ setAddress(String):void
+ getBankID(): String
+ checkBankID(String):String
<<Interface>>
ILogic
+ injectLink(ILink):void
+ startConnection():void
+ sendMessage(String):void
+ receiveMessage():String
+ login(String, String):String
+ toProtocol03(String, String, String, String):String
+ getCustomer():ICustomer
+ getAdmin():IAdmin
+ toProtocol07(String, String, String, String, String, String, String):String
+ getAccountBalance(String):int
+ logout():String
+ toProtocol05(String, String, String, String, String):String
+ getTransactionHistory(String, String):String
+ sendMailAutogenerated(String, String, String, String):String
+ toProtocol09(String, boolean):String
+ updatePassword(String, String, String):String
+ toProtocol17(String):String
+ checkPassword(String, String):String
+ concatcBank(String, String, String):String
+ openBankAccount():String
+ changeTransactionCategory(String, String, String):String
+ lastLogin():String
<<Interface>>
ILink
+ startConnection():void
+ sendMessage(String):void
+ receiveMessage():String
+ endConnection():void
<<Interface>>
IUser
+ getID():String
+ getEmail():String
+ setEmail(String):void
+ getPhoneNo():String
+ setPhoneNo(String):void

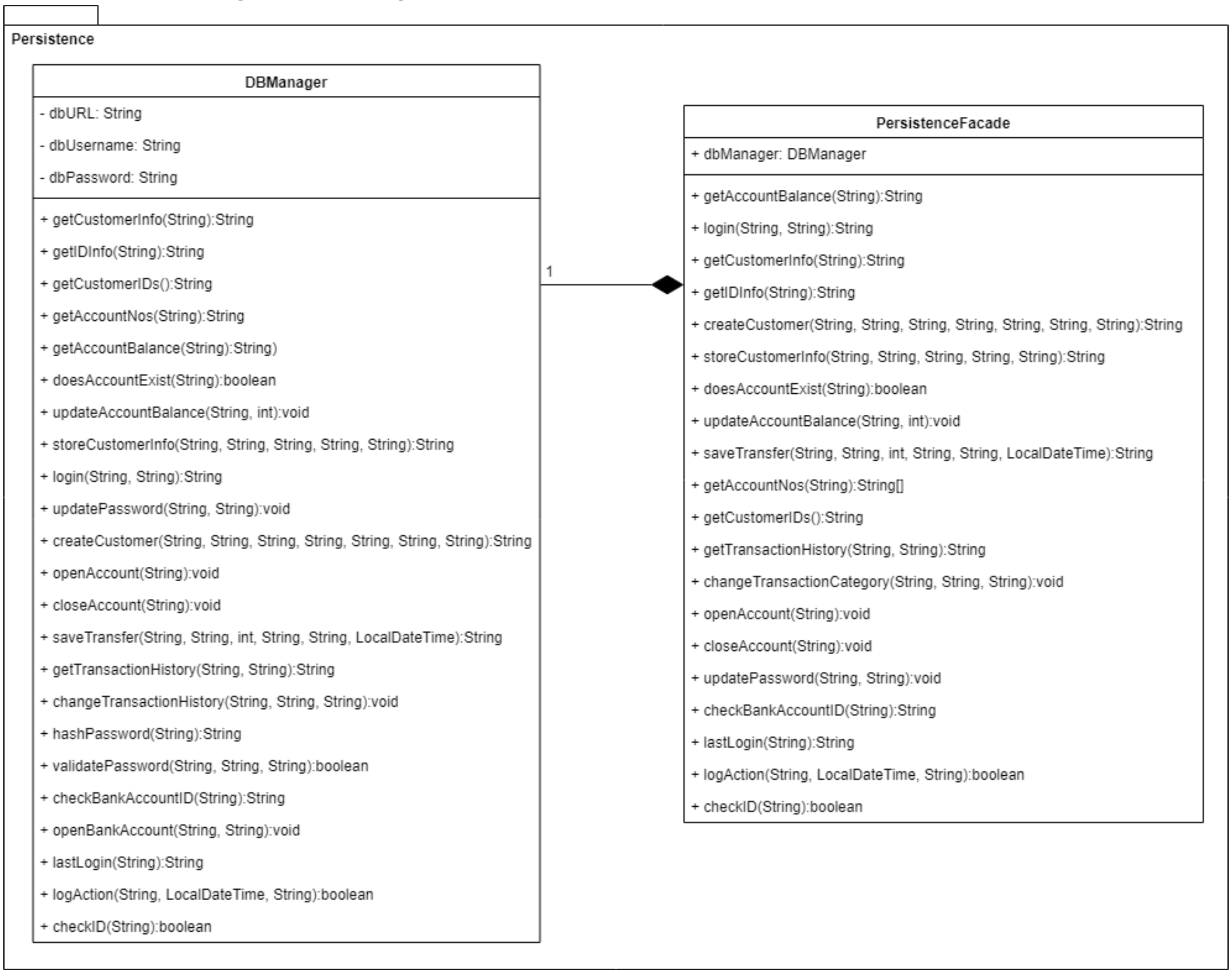
Web-service design class diagram - Link


Link
ClientConnection
+ port: int
+ algorith: String
- server: ServerSocket
- SSLserver: SSLServerSocket
- logic: ILogic
+ ClientConnection(String, ILogic)
+ establishCommunication():void
+ getSocketAddress():InetAddress
+ getPort():int
1
LinkFacade
- logic: ILogic
- connection: ClientConnection
+ injectLogic(ILogic):void
+ startConnection():void
*
HandleConnection
- SSLSocket: SSLSocket
- logic: ILogic
+ HandleConnection(SSLSocket, ILogic)
+ run():void

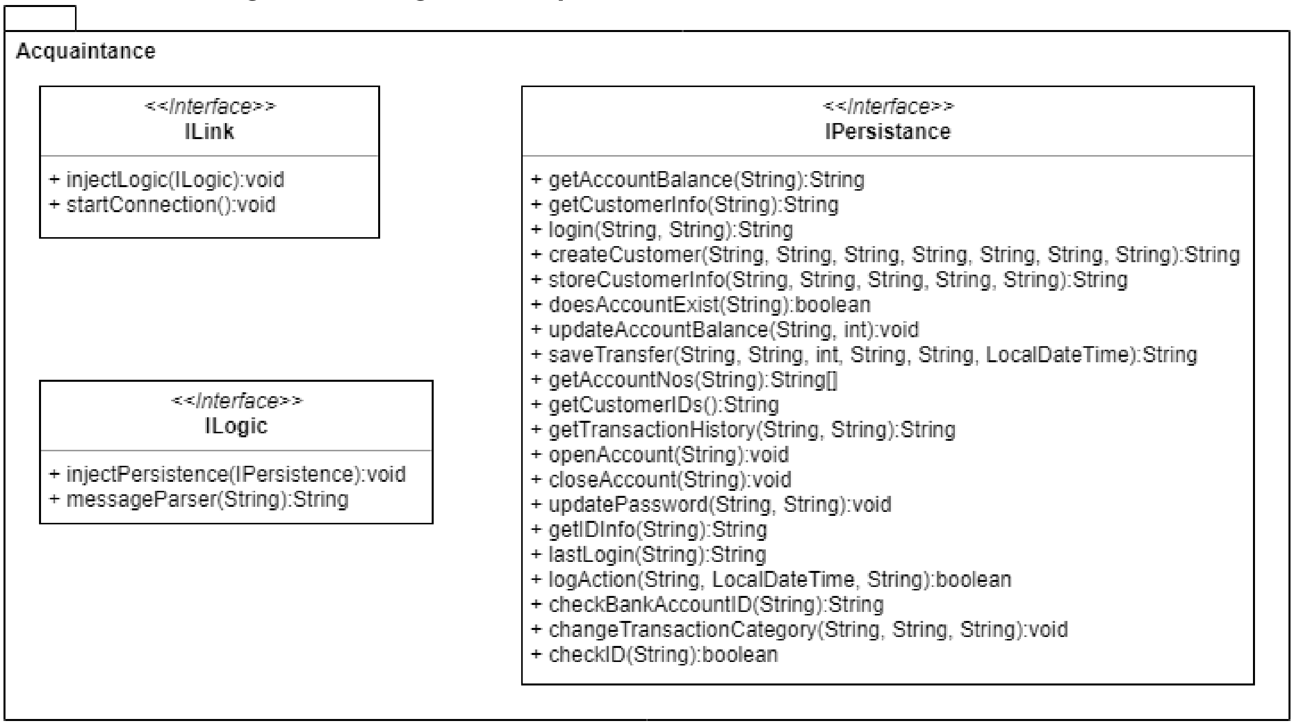
Web-service design class diagram - Logic


Logic
messageParser
- serverProtocol: ServerProtocol
+ MessageParser()
+ fromProtocol(String):String
1
LogicFacade
- messageparser: MessageParser
- persistence: IPersistence
+ injectPersistance(IPersistence)
+ messageParser(String):String
Transfer
- persistence: IPersistence
- fromAccount: String
- toAccount: String
- amount: int
- category: String
- text; String
- date: LocalDateTime
- customerID: String
+ Transfer(String, String, String, String, IPersistence, String, String)
+ validate():String
+ isAccount(String):boolean
+ completeTransfer():String
+ saveTransfer():String
0..*
1
ServerProtocol
- persistence: IPersistence
- mailHandler: MailHandler
- logger: ActionLogger
+ ServerProtocol(IPersistence)
+ serverHandler(String[]): String
1
MailHandler
- username: String
- password: String
+ sendMailAutogenerated(String, String, String, String):String
+ sendResetPasswordAutogenerated(String, String, String):String
+ contactBankMail(String, String, String, String):String
1
ActionLogger
- persistence: IPersistence
+ ActionLogger(IPersistence)
+ logAction(String, String):boolean

Web-service design class diagram - Persistence


Persistence
DBManager
- dbURL: String
- dbUsername: String
- dbPassword: String
+ getCustomerInfo(String):String
+ getIDInfo(String):String
+ getCustomerIDs():String
+ getAccountNos(String):String
+ getAccountBalance(String):String)
+ doesAccountExist(String):boolean
+ updateAccountBalance(String, int):void
+ storeCustomerInfo(String, String, String, String, String):String
+ login(String, String):String
+ updatePassword(String, String):void
+ createCustomer(String, String, String, String, String, String, String):String
+ openAccount(String):void
+ closeAccount(String):void
+ saveTransfer(String, String, int, String, String, LocalDateTime):String
+ getTransactionHistory(String, String):String
+ changeTransactionHistory(String, String, String):void
+ hashPassword(String):String
+ validatePassword(String, String, String):boolean
+ checkBankAccountID(String):String
+ openBankAccount(String, String):void
+ lastLogin(String):String
+ logAction(String, LocalDateTime, String):boolean
+ checkID(String):boolean
1
PersistenceFacade
+ dbManager: DBManager
+ getAccountBalance(String):String
+ login(String, String):String
+ getCustomerInfo(String):String
+ getIDInfo(String):String
+ createCustomer(String, String, String, String, String, String, String):String
+ storeCustomerInfo(String, String, String, String, String):String
+ doesAccountExist(String):boolean
+ updateAccountBalance(String, int):void
+ saveTransfer(String, String, int, String, String, LocalDateTime):String
+ getAccountNos(String):String[]
+ getCustomerIDs():String
+ getTransactionHistory(String, String):String
+ changeTransactionCategory(String, String, String):void
+ openAccount(String):void
+ closeAccount(String):void
+ updatePassword(String, String):void
+ checkBankAccountID(String):String
+ lastLogin(String):String
+ logAction(String, LocalDateTime, String):boolean
+ checkID(String):boolean

Web-service design class diagram - Acquaintance


Acquaintance
<<Interface>>
ILink
+ injectLogic(ILogic):void
+ startConnection():void
<<Interface>>
ILogic
+ injectPersistence(IPersistence):void
+ messageParser(String):String
<<Interface>>
IPersistance
+ getAccountBalance(String):String
+ getCustomerInfo(String):String
+ login(String, String):String
+ createCustomer(String, String, String, String, String, String, String):String
+ storeCustomerInfo(String, String, String, String, String):String
+ doesAccountExist(String):boolean
+ updateAccountBalance(String, int):void
+ saveTransfer(String, String, int, String, String, LocalDateTime):String
+ getAccountNos(String):String[]
+ getCustomerIDs():String
+ getTransactionHistory(String, String):String
+ openAccount(String):void
+ closeAccount(String):void
+ updatePassword(String, String):void
+ getIDInfo(String):String
+ lastLogin(String):String
+ logAction(String, LocalDateTime, String):boolean
+ checkBankAccountID(String):String
+ changeTransactionCategory(String, String, String):void
+ checkID(String):boolean

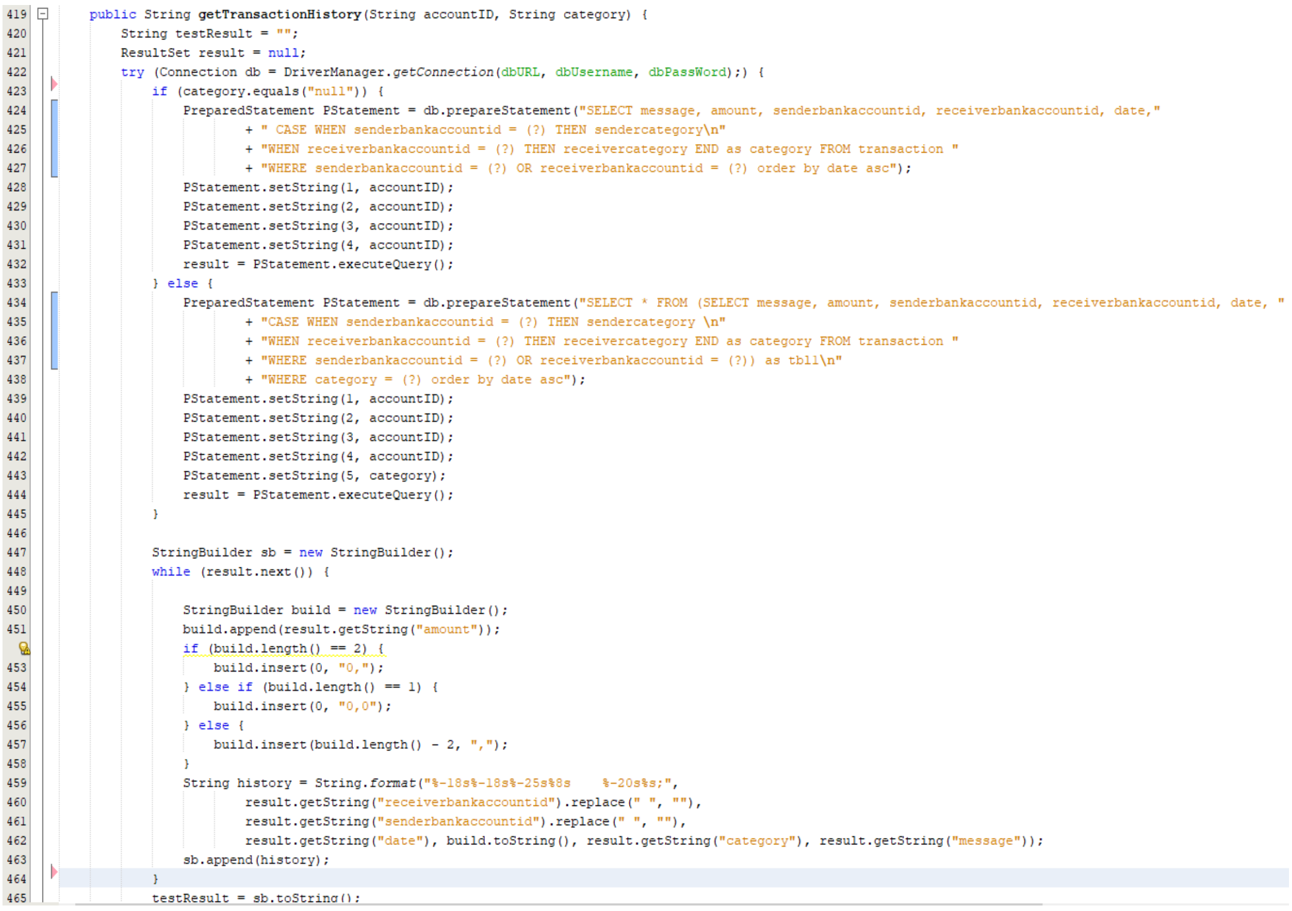
Appendix E - Project log

Date	Activities
21-09-2018	We set up Trello and made the first draft for the project foundation. We assigned roles to the people in the group and we chose to use Scrum
29-09-2018	We made changes in the project foundation document in connection with the supervisor meeting. some of the changes is: changes requirements and made new sections. we wrote an email to group 9 and asked when they could attend the review meeting. Our project foundation was handed in at 22 o' clock.
05-10-2018	We went through the notes from the review meeting. Today we looked at the requirements for the project and specified them. We ranked the requirements with MoSCoW. We planned the different things that we should change in the document. We assigned tasks to each group member.
12-10-2018	Today we corrected and changed the last few things in the project foundation. We handed the foundation document in. We made the project backlog and defined the sprint backlog. we have talked about how to make the connection between the client and the server
22-10-2018	We began with a sprint to make different diagrams for the system. We started on making an E/R diagram, analysis class diagram, detailed use cases, sketch of the GUI etc.
23-10-2018	Today we reviewed our different diagrams and made interaction diagrams. We attempted to make a connection with our java server program. We also connected to the ubuntu server we got from Ronald.
24-10-2018	Today we finished reviewing the diagrams. We made changes to the E/R diagram and made the database. We made code that connected the database to our web-service. We started the implementation of the protocol for connections between the client and web-service.
25-10-2018	Today we continued with the programming of the different subsystems in the system. We also made set up the report so it is ready to be written in. We have made the GUI fxml files but not yet committed it to the master branch.
26-10-2018	Today we finished a lot of the functionality for the web-service. View account balance, login, and create customer have all been finished for the server. The GUI have pretty much been finished at this point, with only minor tweaking needed for it to be completely done. Overall most of the day have been spent coding, and committing finished code to the master branches.
02-11-2018	Today we finished the sprint and planned the next sprint. We had a meeting about the finished sprint - about what was good and bad
07-11-2018	Today we helped some group members that had been absent, get introduced to how far we have coded, and how the code works, after that, people began

	choosing from the sprint backlog, and began working.
09-11-2018	Today we coded and merged different branches into master. The video was made and handed in.
14-11-2018	Today we continued with coding.
16-11-2018	Today we reviewed group 11’s video and handed the review in. Today we coded and merged different branches into master. We started a new sprint. We were only 4 people today.
21-11-2018	Today the encryption was made and merged to master. Some people in the group started writing on the report and others continued with the programming.
23-11-2018	Today we removed session from the code so we can have multiple users on the same server. We also cleaned up some of the printouts to the console. We also started on sanitizing the input to the system. There was also continued work on writing the report.
28-11-2018	Today we continued working on the code and the report. We had a meeting with our supervisor. We planned a new sprint and talked about how the group was doing and what went well in the sprint and what was bad. We decided to change our start time - We agreed to start at 9:15 sharp, because some people were often late. at 9:15 all people should be ready and have their pc ready for the scrum meeting.
30-11-2018	We started working on the things we planned at the end of last time. Today we mostly coded (new features and bug fixes). We also merged some code into master before we started coding today.
05-12-2018	Today we coded some of the last things. The logger is now implemented. Contact bank is also implemented, and a lot of bug fixes. We made a welcome screen. other things has been worked on but not yet implemented.
07-12-2018	Today we implemented the last things in the code. We tested the “web-service” code on the server. We made a code freeze and put the finished “web-service” code on the server We planned the next week, where we plan to write the rest of the report.
11-12-2018	Today we started writing the sections: Design, implementation, Testing and more in the report.
12-12-2018	Today we continued writing on the report. We had a meeting with our supervisor.
13-12-2018	Today we continued writing on the report, but we worked at home, individually (but communicated over the internet). We updated a lot of the diagrams. Most of the text for the report is done, but needs to be reviewed and corrected by all the group members.
14-12-2018	Today we wrote the last things for the report and began to review and correct the report.

15-12-2018	Today we made further reviews and corrections to the report
16-12-2018	Today we went through the report and corrected the last things in the report, to prepare for handing in.

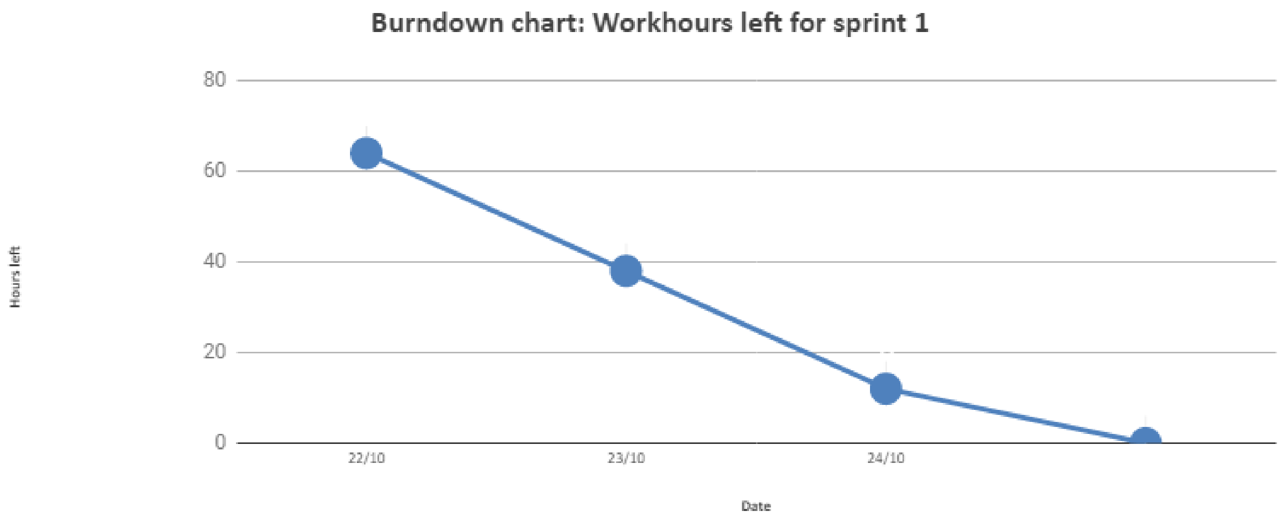
Appendix F - Transaction history



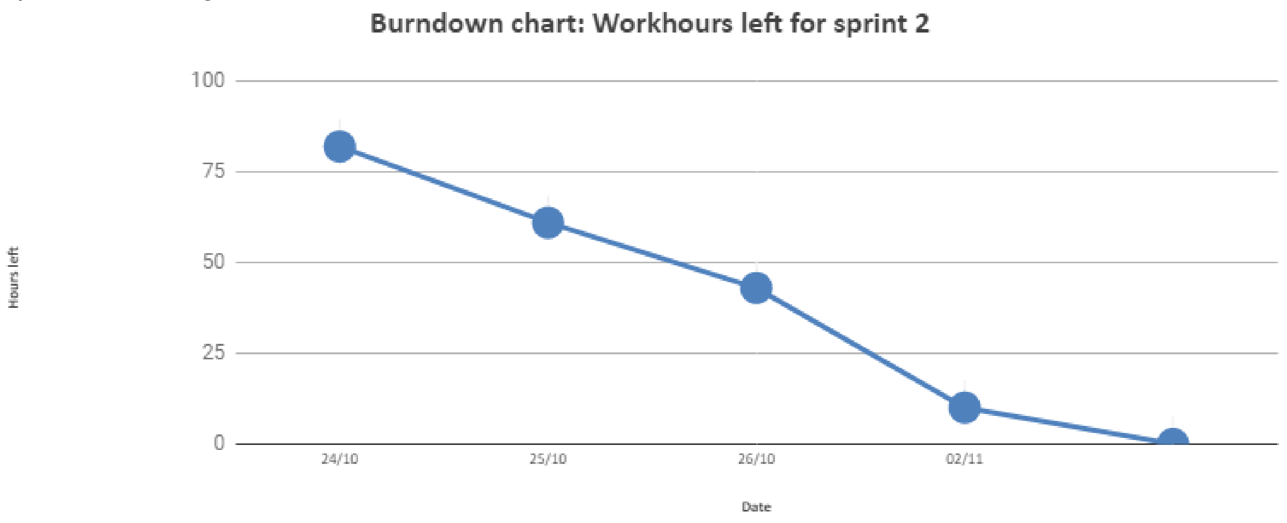
```java
    public String getTransactionHistory(String accountID, String category) {
        String testResult = "";
        ResultSet result = null;
        try (Connection db = DriverManager.getConnection(dbURL, dbUsername, dbPassWord);) {
            if (category.equals("null")) {
                PreparedStatement PStatement = db.prepareStatement("SELECT message, amount, senderbankaccountid, receiverbankaccountid, date,"
                        + " CASE WHEN senderbankaccountid = (?) THEN sendercategory\n"
                        + "WHEN receiverbankaccountid = (?) THEN receivercategory END as category FROM transaction "
                        + "WHERE senderbankaccountid = (?) OR receiverbankaccountid = (?) order by date asc");
                PStatement.setString(1, accountID);
                PStatement.setString(2, accountID);
                PStatement.setString(3, accountID);
                PStatement.setString(4, accountID);
                result = PStatement.executeQuery();
            } else {
                PreparedStatement PStatement = db.prepareStatement("SELECT * FROM (SELECT message, amount, senderbankaccountid, receiverbankaccountid, date, "
                        + "CASE WHEN senderbankaccountid = (?) THEN sendercategory \n"
                        + "WHEN receiverbankaccountid = (?) THEN receivercategory END as category FROM transaction "
                        + "WHERE senderbankaccountid = (?) OR receiverbankaccountid = (?)) as tbl1\n"
                        + "WHERE category = (?) order by date asc");
                PStatement.setString(1, accountID);
                PStatement.setString(2, accountID);
                PStatement.setString(3, accountID);
                PStatement.setString(4, accountID);
                PStatement.setString(5, category);
                result = PStatement.executeQuery();
            }

            StringBuilder sb = new StringBuilder();
            while (result.next()) {

                StringBuilder build = new StringBuilder();
                build.append(result.getString("amount"));
                if (build.length() == 2) {
                    build.insert(0, "0,");
                } else if (build.length() == 1) {
                    build.insert(0, "0,0");
                } else {
                    build.insert(build.length() - 2, ",");
                }
                String history = String.format("%-18s%-18s%-25s%8s    %-20s%s;",
                        result.getString("receiverbankaccountid").replace(" ", ""),
                        result.getString("senderbankaccountid").replace(" ", ""),
                        result.getString("date"), build.toString(), result.getString("category"), result.getString("message"));
                sb.append(history);
            }
            testResult = sb.toString();
```

Appendix G - Burndown charts

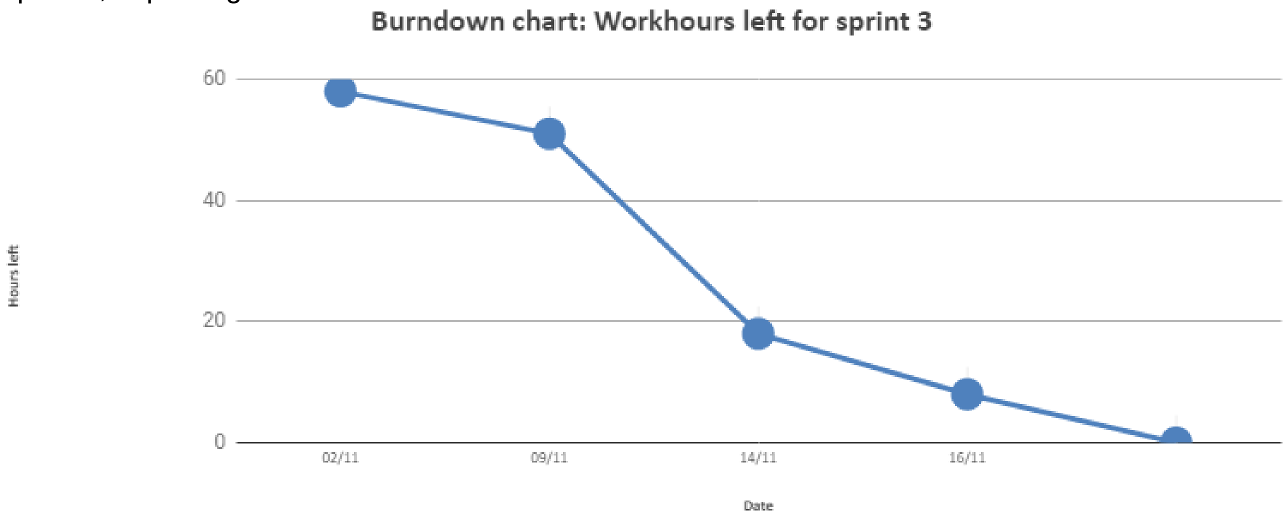
Sprint 1, project startup



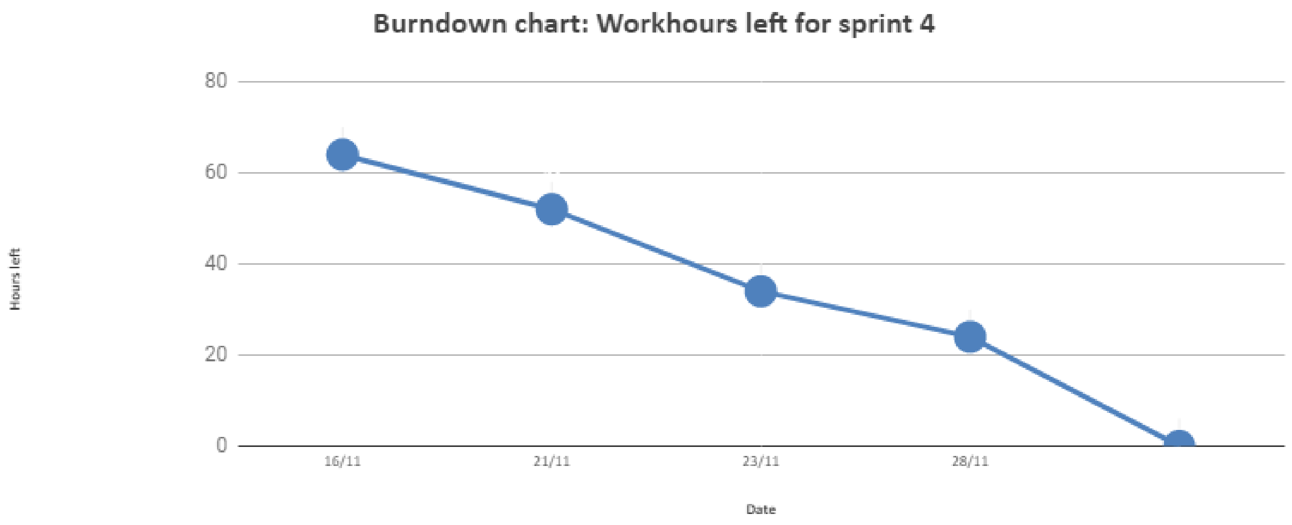
Sprint 2, building code base



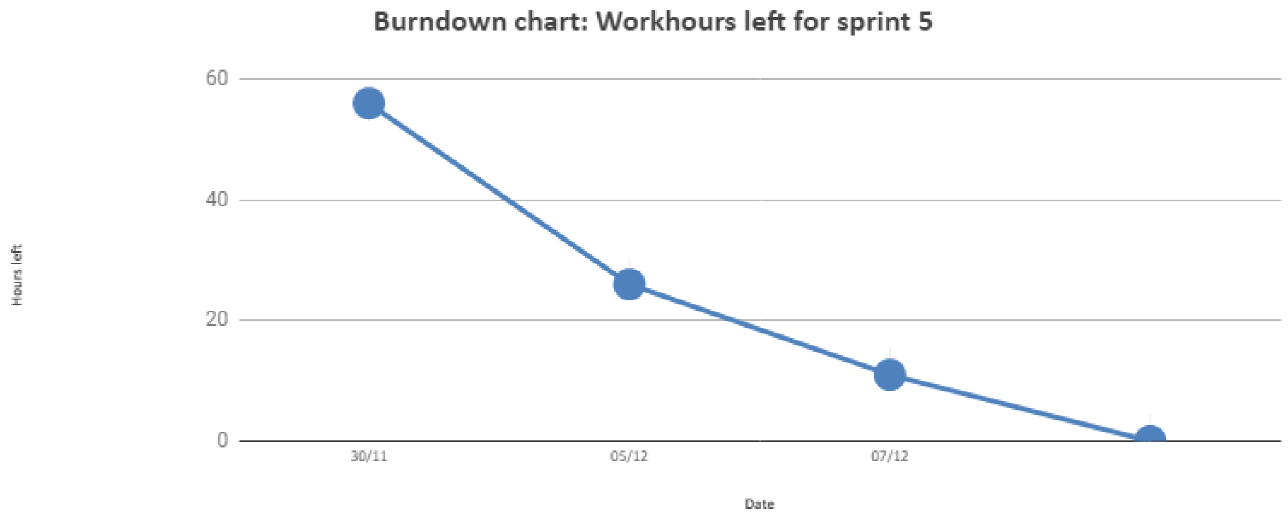
Sprint 3, improving code base



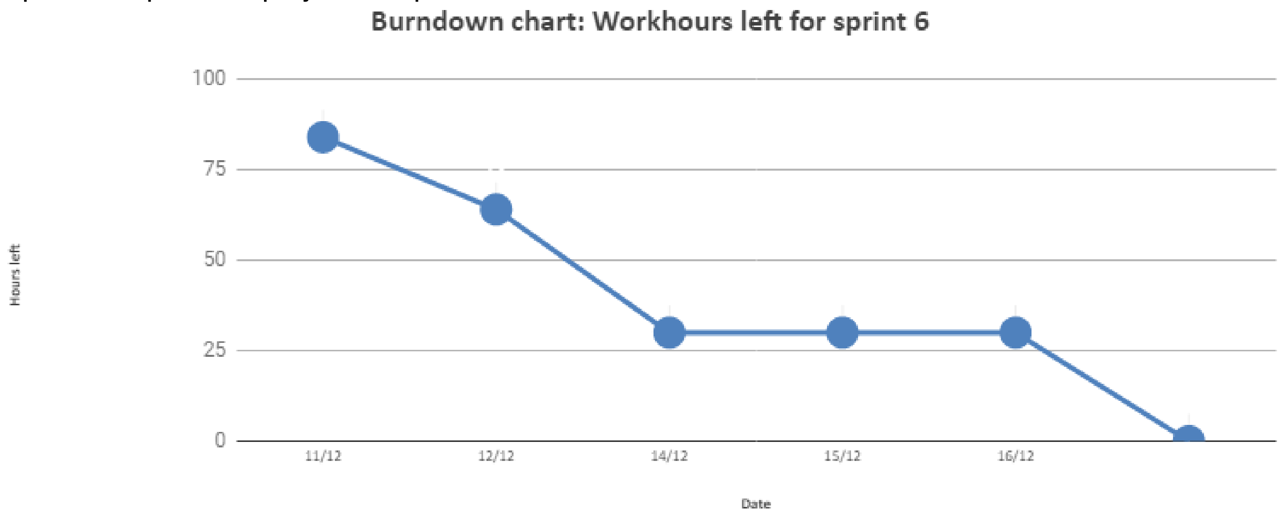
Sprint 4, Start work on report, finish high priority code



Sprint 5, code completion



Sprint 6, report and project completion


Burndown chart: Workhours left for sprint 6
Hours left
11/12
12/12
14/12
15/12
16/12
Date