<Banking Web Application>
Analysis and Design Document

Student: Bogdan Stupariu
Group: 30432

Table of Contents

1. Requirements Analysis

1.1 Assignment Specification

[Application description]

This assignment requests to design and implement an application for the front desk employees of a bank and has as main objective making students familiar with architectural patterns. The application should have two types of users: a regular one represented by the front desk employee whose main attributes will be to perform operations on clients and their accounts, and the administrator, whose main prerogatives will be to perform operations on the regular users and also monitories their activity.

1.2 Functional Requirements

[Present the functional requirements]

The regular user should have the following possibilities: operate CRUD on the clients and their accounts, transfer money between accounts and also process utilities bills. As described before, the administrator will have the possibility to perform CRUT operations on regular users and also generate reports for a particular period containing the activities performed by an employee. From those, there can be obviously deducted that the application should have a login functionality. All the data should be validated against invalid data before submitting and so a response functionality for input data should also be required. Also, the reports should be generated and for better using the data it should be generated in a pdf file also.

In most of the sides of the application, the input given by the user (each type) should generate a response as output, which will be generated in tables organizing data from the database in a readable and pleasurable way.

Another important side when it comes about a banking application, will be security. It is required that at least there is no possibility of directly accessing the links without being logged in or even not accessing directly any other link than the index one (that is redirecting to the login page).

1.3 Non-functional Requirements

[Discuss the non-functional requirements for the system]

The non-functional basic requirements are the usage of the layer's architectural pattern for application organization, usage of a domain logic pattern (transaction script or domain model) / a data source hybrid pattern (table module, active record) and a data source pure pattern (table data gateway, row data gateway, data mapper). Also, all the data should be stored in a database.

Accessibility should be provided by both implementing the web application as an online application with the obvious restriction of needing an internet connection, and also giving the possibility of using the application offline, by individually installing all the components on the computer.

2. Use-Case Model

[Create the use-case diagrams and provide one use-case description (according to the format below).

Use-Case description format:

Use case: <use case goal>

Level: <one of: summary level, user-goal level, sub-function>

Primary actor: <a role name for the actor who initiates the use case>

Main success scenario: <the steps of the main success scenario from trigger to goal delivery>

Extensions: <alternate scenarios of success or failure>

]

Use case:

Make a transaction between accounts

Level:

user-goal level – directly requested by the regular user with the goal of moving an amount of money form one

account to another. Response is directly sent to the same user.

Primary actor:

Regular user logged in in the web application that launches the process of transaction by filling the requested inputs and pressing the requested button.

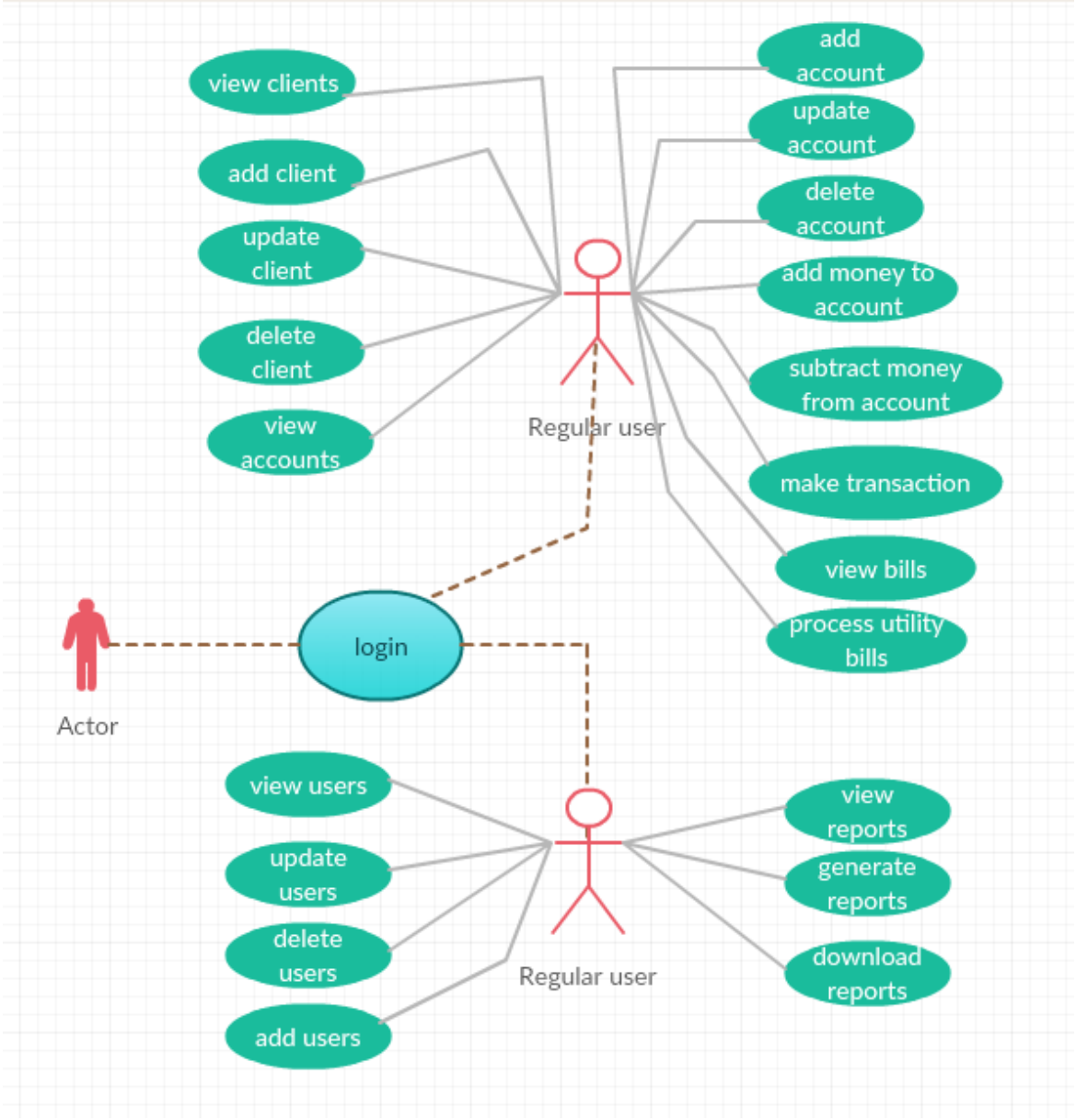
Main success scenario:

The button is triggered and the data validation process begins. The input date is validated and a request is made to the backend in order to verify if there is no illegal transaction such as transaction to itself, transacting more than the actual amount in account. After all verification tests are passed, a request is made to the back end and then both the sending account and receiving account are updated – the before credit amount is updated (receiver +, sender -).

Extensions:

The data input is not well formatted and a message will appear on the desktop. May happen if the amount to be send is not a proper one (e.g. not only numerical characters, not in the requested xx.xx format), account to be send in is not chosen, or amount sent is greater than the amount in the account.

The receiving account may be an existing one in the database (like being an account administrated by the same bank) and then the amount transferred will remain the bank. Also it may not be in the database, and then the amount is not in the administrated database anymore – only a substraction.



3. System Architectural Design

3.1 Architectural Pattern Description

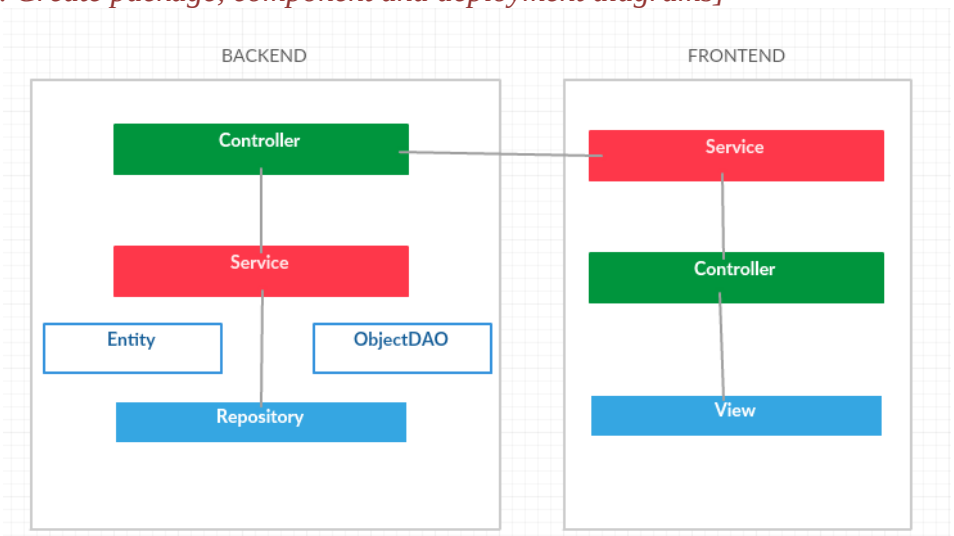
[Describe briefly the used architectural patterns.]

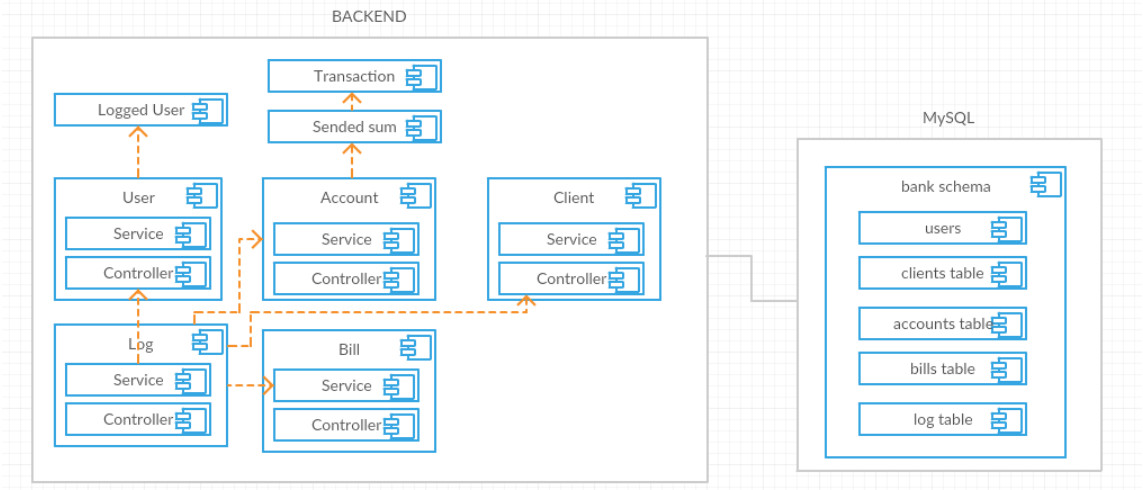
Layered Architecture - Partitions the concerns of the application into stacked groups (layers). Usage in the backend of Controllers, Services, Entities and repositories. Usage in the frontend of Views, Services and Controllers.
Domain Business logic patters - An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain. Usage of the entities in the backend and making the business logic in the services side.

Table Data Gateway - The first Data Source pattern is the Table Data Gateway. This pattern is sometimes also referred to as a Data Access Object or Data Access Component (although that terminology is also used to refer to any class used in the Data Access Services layer). Basically, the idea is that the class encapsulates the SQL (statements or stored procedures) for accessing sets of related tabular data in a database. These sets can be queried from multiple tables, views, or joins and so needn't, and typically won't, map to single tables.

3.2 Diagrams

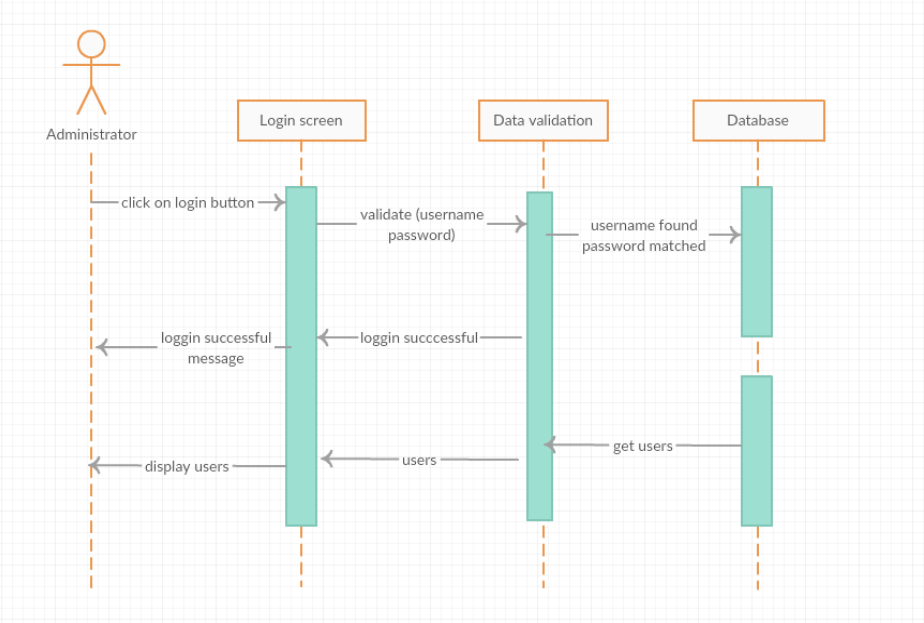
[Create the system's conceptual architecture; use architectural patterns and describe how they are applied. Create package, component and deployment diagrams]





4. UML Sequence Diagrams

[Create a sequence diagram for a relevant scenario.]



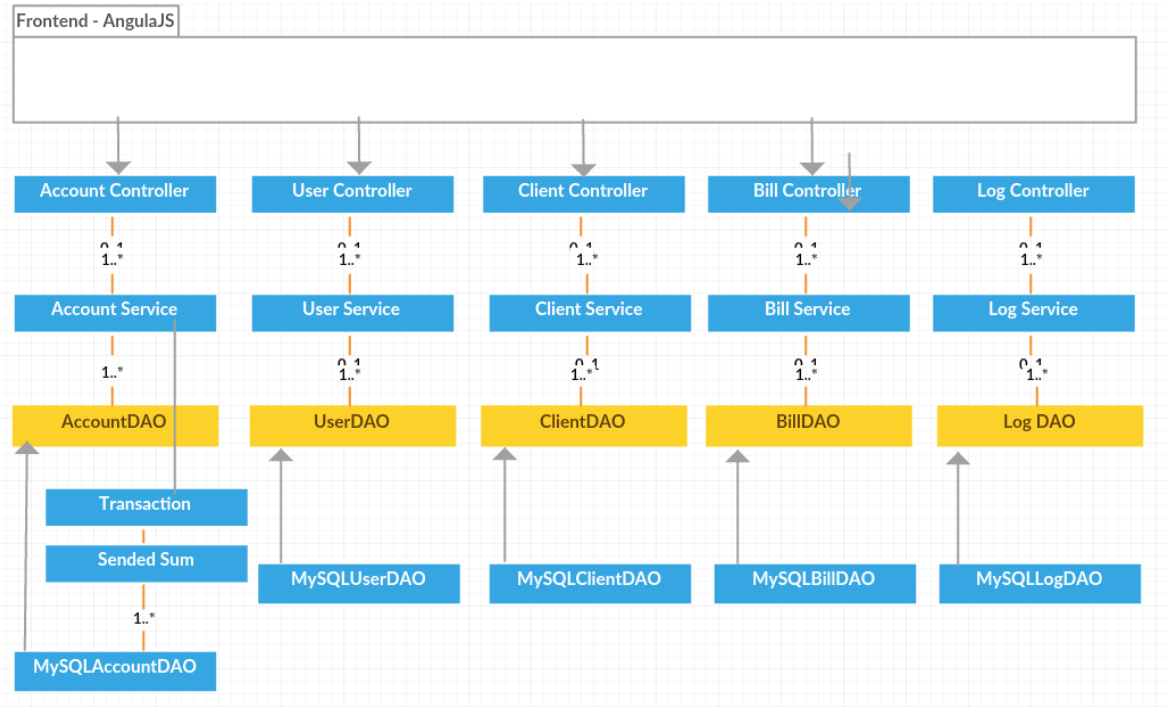
5. Class Design

5.1 Design Patterns Description

[Describe briefly the used design patterns.]

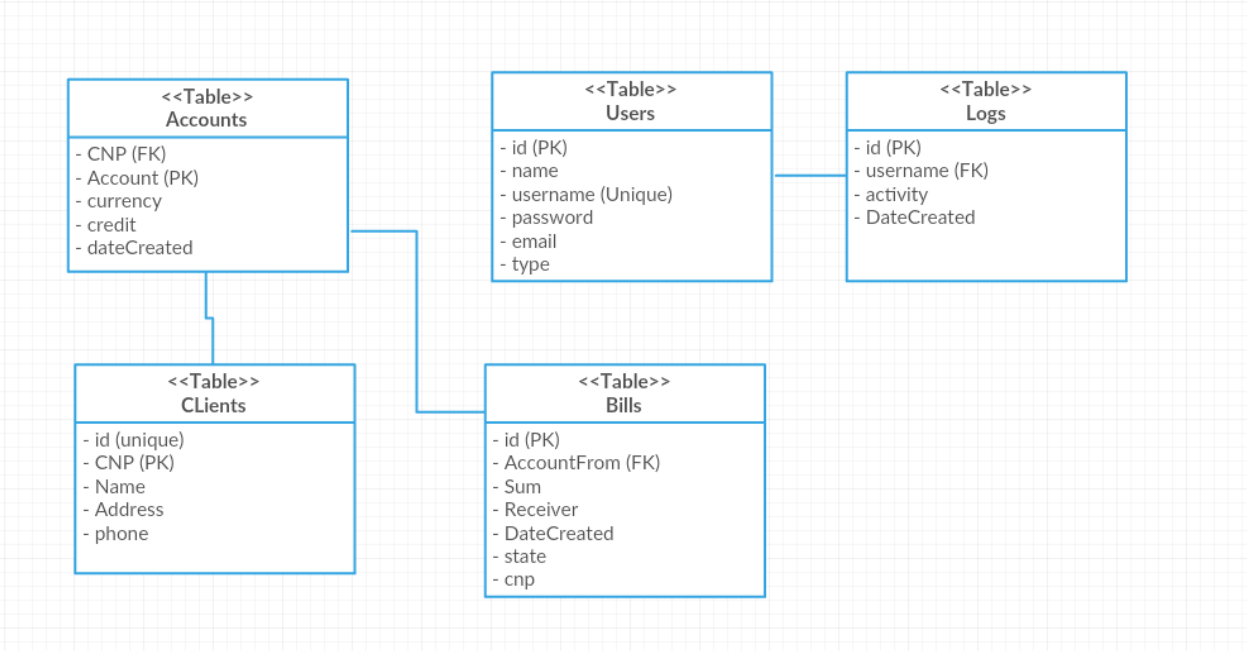
5.2 UML Class Diagram

[Create the UML Class Diagram and highlight and motivate how the design patterns are used.]



6. Data Model

[Present the data models used in the system's implementation.]



7. System Testing

[Present the used testing strategies (unit testing, integration testing, validation testing) and testing methods (data-flow, partitioning, boundary analysis, etc.).]

All input data is tested against invalid data by using regex data validation. Regex Pattern are supposed to be matching the input data and so to validate the input. Otherwise, the input will be rejected and a message will occup.
Examples of patterns:
CNP Validation:

```
var pattern = /^(1|2|5|6)(0[0-9]|1[0-7]|[3-9][0-9])(0[1-9]|[10]|[11]|[12])([0-2][1-9]|[20]|3[0-1])\d{6}$/im;
```

IBAN Validation:

```
var pattern = /^([R][O]|[E][U]|[U][S])[0-9][0-9] [S][B][B][K] ([0-9]|[A-Z]){4} [0-9]{4} [0-9]{4} [0-9]{4}$/im;
```

Date Validation:

```
var pattern = /^20[0-1][0-9]\-((0[1-9])|(1[0-2]))\-((0[1-9])|((1[0-9])|(2[0-9])|(3[0-1])))$/im;
```

8. Bibliography

1. http://stackoverflow.com/
2. https://docs.angularjs.org/api
3. https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
4. http://getbootstrap.com/components/
5. https://www.codecademy.com/learn/javascript
6. Youtube tutorials