Final Graduation Project Documentation

Mohamed Magdy Dewidar

ID: 2712072401

Abstract:

This project focuses on developing a Bank Management System using the client-server model with the Qt/C++ framework. The system comprises a client-side GUI application for user interaction and a server-side console application for processing requests. It supports two types of users: Standard Users and Admin Users, with role-based access control ensuring secure and structured banking operations.

Standard users can log in, view their account balance, check transaction history, and perform fund transfers. Admin users are granted extended privileges such as creating, updating, deleting users, and viewing the entire bank database. Client-server communication is facilitated through Sockets, and all requests are encrypted for secure transmission over the network. The server is designed to handle multiple requests simultaneously via multi-threading, ensuring efficient performance.

The GUI provides an intuitive user experience by formatting structured data, such as transaction history, into readable tables. Meanwhile, the server logs client requests for auditing and debugging and can optionally run as a system service for continuous background operation.

This project demonstrates essential concepts such as multi-threading, encryption, and role-based access control, resulting in a secure, scalable, and user-friendly platform for managing banking operations.

Introduction

The modern banking industry demands secure, efficient, and user-friendly systems to manage financial operations. This project aims to create a Bank Management System that utilizes the client-server model to deliver a distributed platform for core banking functionalities. Built using the Qt/C++ framework, the system ensures role-based authorization for both standard users and administrators, addressing their distinct needs.

The system comprises two applications: a client-side GUI for intuitive user interaction and a server-side console application for processing requests. Secure communication between the two is achieved using Sockets, with data encrypted during transmission to protect sensitive information such as account details and transaction records.

Key features include operations such as logging in, viewing account balances, performing transactions, and retrieving transaction histories for standard users. Administrators have extended functionalities, including managing user accounts and accessing the bank database. To ensure high performance and scalability

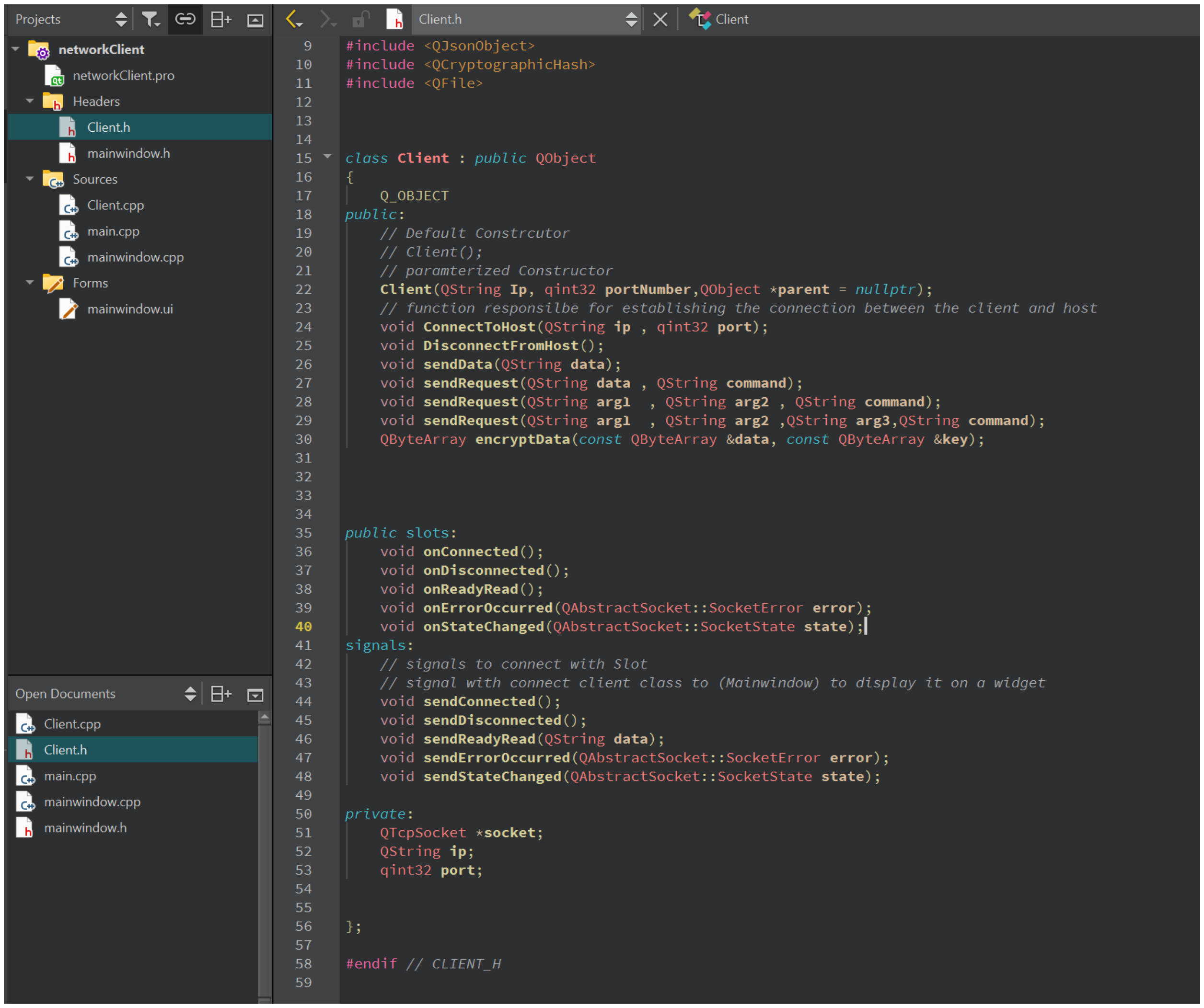
The client application presents data in a clean, organized table format, enhancing user readability. The server further logs all client requests for auditing and monitoring purposes, with the option to configure it as a system service for seamless background operation.

This project integrates secure communication, scalable architecture, and role-based access control to create a robust and reliable banking system. By implementing these features, it demonstrates practical solutions to real-world challenges in banking operations.

Implementation:

Client Side:

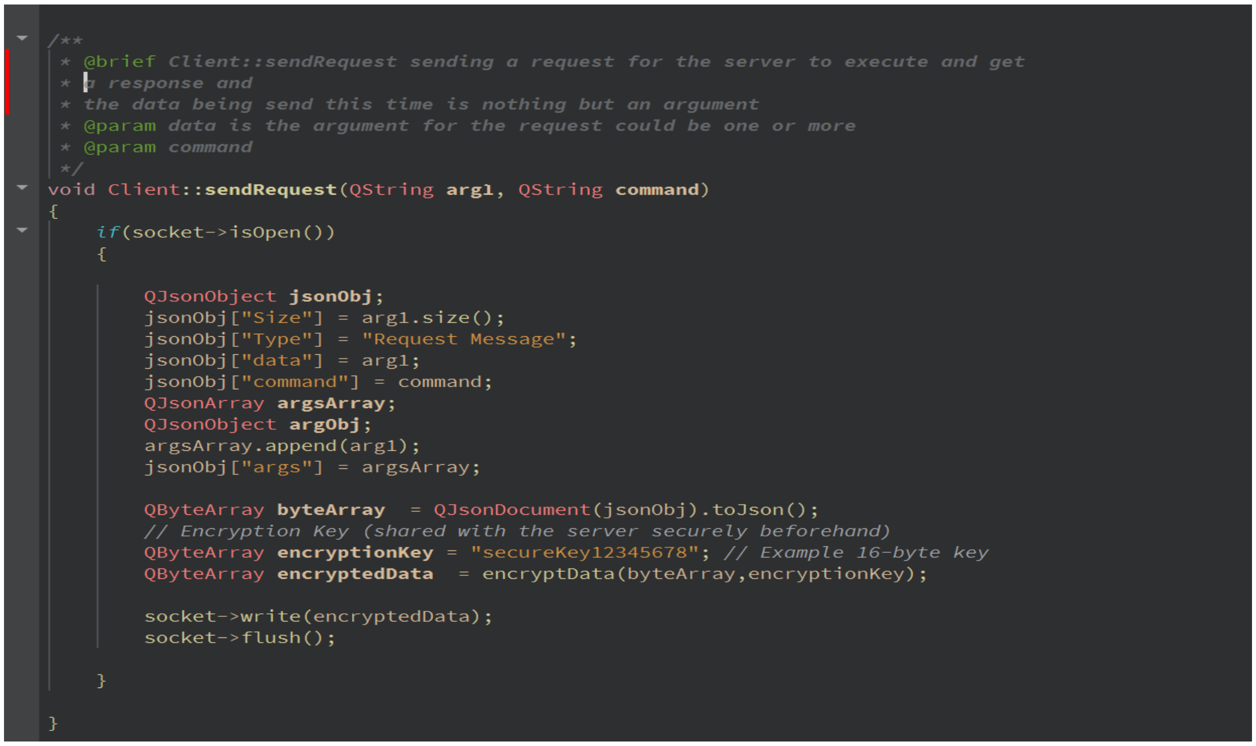
Firstly, will Break down how the communication system between client and server was built and what features does it offer.



```cpp
#include <QJsonObject>
#include <QCryptographicHash>
#include <QFile>



class Client : public QObject
{
    Q_OBJECT
public:
    // Default Constrcutor
    // Client();
    // paramterized Constructor
    Client(QString Ip, qint32 portNumber,QObject *parent = nullptr);
    // function responsilbe for establishing the connection between the client and host
    void ConnectToHost(QString ip , qint32 port);
    void DisconnectFromHost();
    void sendData(QString data);
    void sendRequest(QString data , QString command);
    void sendRequest(QString arg1  , QString arg2 , QString command);
    void sendRequest(QString arg1  , QString arg2 ,QString arg3,QString command);
    QByteArray encryptData(const QByteArray &data, const QByteArray &key);




public slots:
    void onConnected();
    void onDisconnected();
    void onReadyRead();
    void onErrorOccurred(QAbstractSocket::SocketError error);
    void onStateChanged(QAbstractSocket::SocketState state);
signals:
    // signals to connect with Slot
    // signal with connect client class to (Mainwindow) to display it on a widget
    void sendConnected();
    void sendDisconnected();
    void sendReadyRead(QString data);
    void sendErrorOccurred(QAbstractSocket::SocketError error);
    void sendStateChanged(QAbstractSocket::SocketState state);

private:
    QTcpSocket *socket;
    QString ip;
    qint32 port;


};

#endif // CLIENT_H
```

The Client class is a core component for managing client-server communication in the banking system using Qt's networking framework. It establishes and maintains connections with the server through **ConnectToHost** and **DisconnectFromHost**, enabling reliable data exchange. The class supports **data transmission** through multiple **sendRequest** methods, allowing flexible communication of commands and parameters. To ensure security, the **encryptData** function encrypts outgoing data, protecting sensitive information during transmission. Where the server will Decrypt the data using Xor encryption.

Event handling is managed through slots like **onConnected**, **onDisconnected**, and **onReadyRead**, which respond to connection status, incoming data, and errors. Custom signals such as **sendConnected** and **sendReadyRead** notify other components, ensuring smooth integration with the client GUI. Built with **QTcpSocket** and Qt's signal-slot mechanism, the Client class provides a secure, modular, and efficient solution for client-server communication, playing a critical role in the system's functionality and usability.

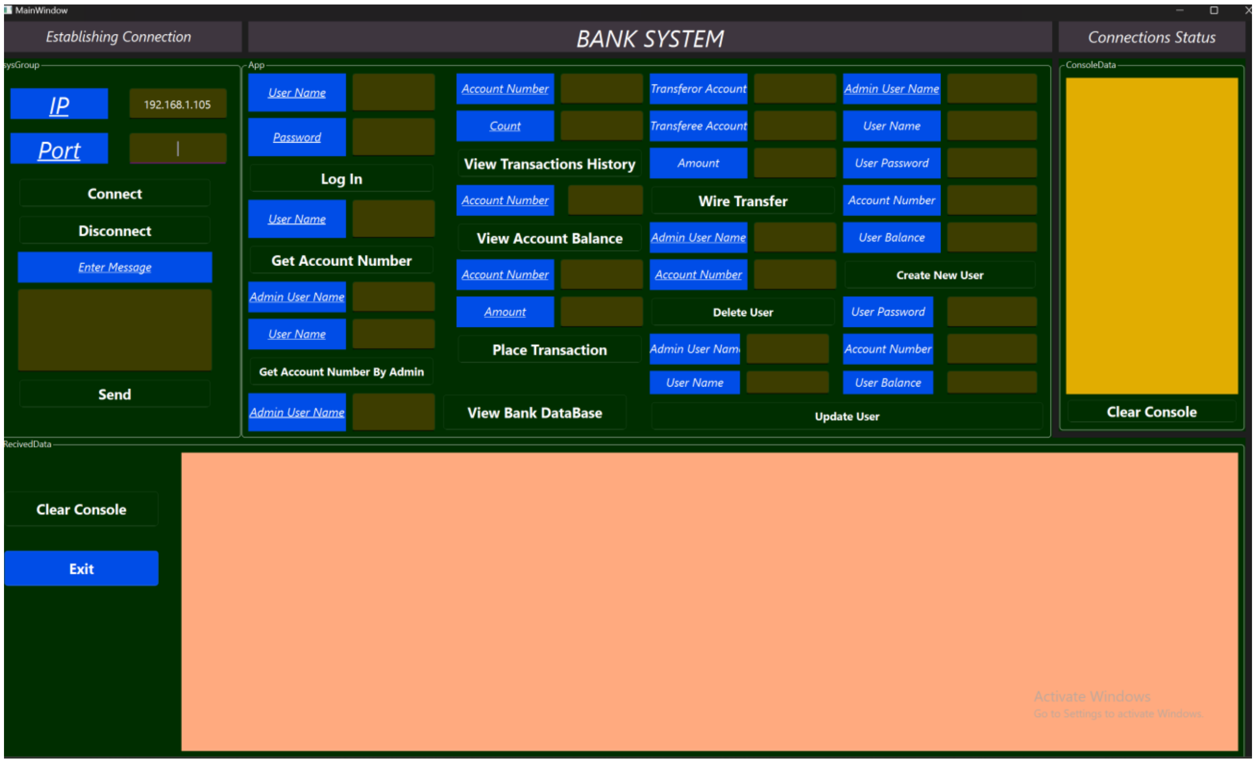


```
/**
 * @brief Client::sendRequest sending a request for the server to execute and get
 * a response and
 * the data being send this time is nothing but an argument
 * @param data is the argument for the request could be one or more
 * @param command
 */
void Client::sendRequest(QString arg1, QString command)
{
    if(socket->isOpen())
    {

        QJsonObject jsonObj;
        jsonObj["Size"] = arg1.size();
        jsonObj["Type"] = "Request Message";
        jsonObj["data"] = arg1;
        jsonObj["command"] = command;
        QJsonArray argsArray;
        QJsonObject argObj;
        argsArray.append(arg1);
        jsonObj["args"] = argsArray;

        QByteArray byteArray  = QJsonDocument(jsonObj).toJson();
        // Encryption Key (shared with the server securely beforehand)
        QByteArray encryptionKey = "secureKey12345678"; // Example 16-byte key
        QByteArray encryptedData  = encryptData(byteArray,encryptionKey);

        socket->write(encryptedData);
        socket->flush();

    }

}
```

The Client class plays a pivotal role in managing data transmission between the client application and the server in the Bank Management System. It ensures secure and organized communication using **JSON format** for request handling. Each **sendRequest** function packages data into a structured JSON object, including metadata such as the **data size**, **type of message**, **command**, and **arguments**. This structured approach mirrors the concept of sending packets with headers, allowing the server to effectively identify the type of request, its content, and associated parameters.

The send Request methods support one, two, or three arguments, offering flexibility for various request types. JSON arrays are used to encapsulate arguments cleanly, ensuring that data remains easy to parse on the receiving end. Once the JSON object is created, it is serialized using **QJsonDocument**, encrypted with a predefined **encryption key**, and transmitted to the server through the **QTcpSocket** connection. The **encryption step** enhances the security of sensitive data during transmission.

By combining **JSON formatting** with encryption, the Client class enables efficient and secure client-server communication. It ensures that requests are structured, traceable, and easily processed, making the overall system robust and reliable for handling complex banking operations.



192.168.1.105

Each time a user enters the valid data and clicks the button a request is sent to server for the server to execute it directly and returns the data to the user through displaying it on a widget. However, the user will have to login to the server first for him to access the features.

Server-Side Implementation:

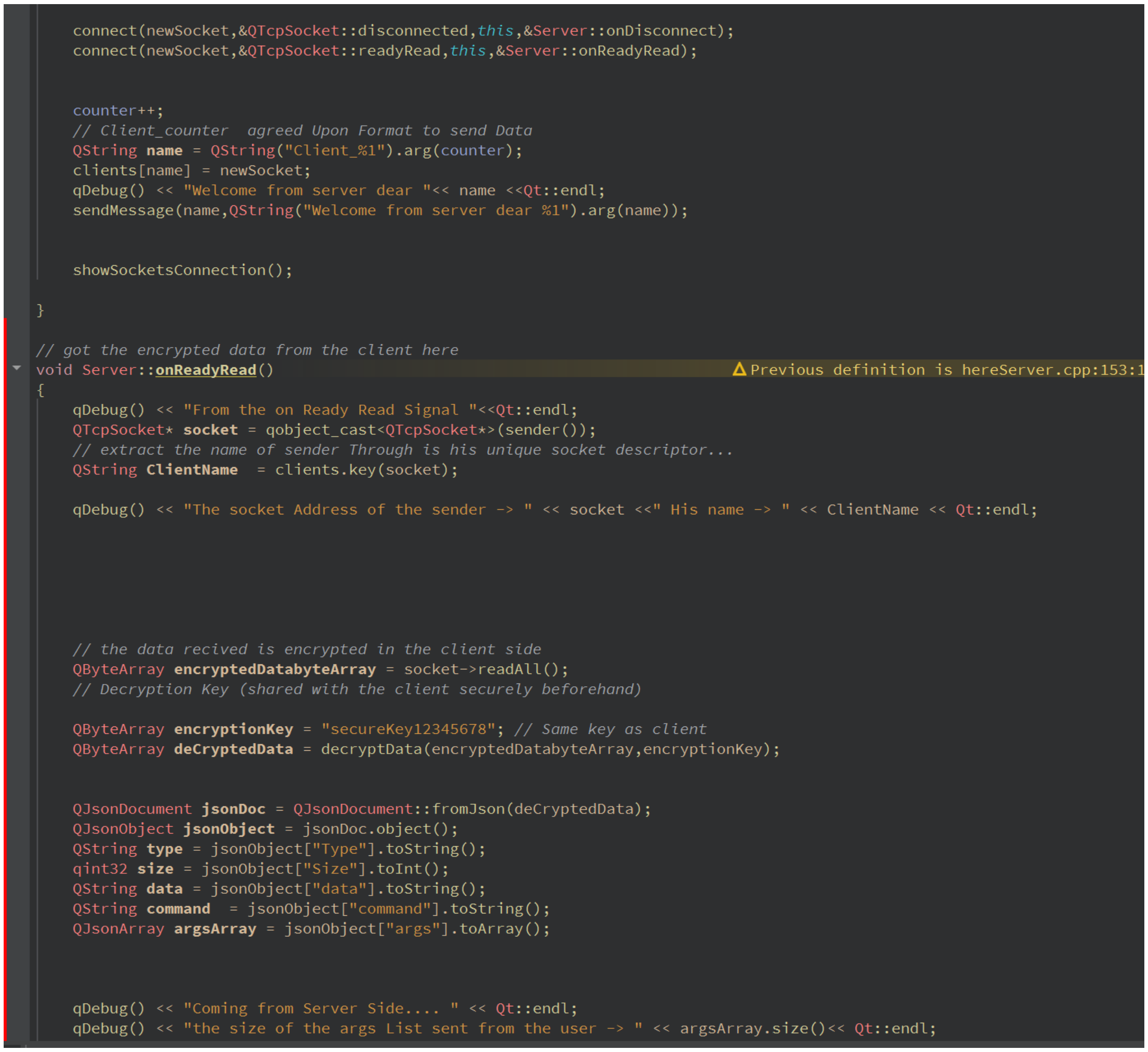
The Server class efficiently handles client connections, secure communication, and data management in a structured and organized way. Every time a client connects to the server, it is assigned a **unique socket descriptor**, which acts as an identifier to send and receive data. In addition, the server assigns each client a **unique name** (e.g., Client_1, Client_2) following an agreed-upon format. This naming convention ensures seamless communication by providing a consistent reference to the client.

To manage client connections, the server uses a **HashMap** (QMap), which maps each client's unique name to its corresponding socket descriptor. This mapping enables the server to easily identify the client receiving the data and handle communication effectively.

The server receives **encrypted data** from the client, which is decrypted using a shared key (secureKey12345678). Once decrypted, the data is parsed in **JSON format** to extract essential fields:

- **Command**: Specifies the operation to be executed for the client.
- **Arguments**: Lists the parameters sent by the client for the command.
- **Size**: Indicates the size of the data to ensure data integrity.

By comparing the received data size with the declared size, the server verifies data integrity and raises warnings if discrepancies are found. This structured approach ensures the server can securely process client requests, extract key information, and execute commands efficiently.



```cpp
    connect(newSocket,&QTcpSocket::disconnected,this,&Server::onDisconnect);
    connect(newSocket,&QTcpSocket::readyRead,this,&Server::onReadyRead);


    counter++;
    // Client_counter  agreed Upon Format to send Data
    QString name = QString("Client_%1").arg(counter);
    clients[name] = newSocket;
    qDebug() << "Welcome from server dear "<< name <<Qt::endl;
    sendMessage(name,QString("Welcome from server dear %1").arg(name));


    showSocketsConnection();

}

// got the encrypted data from the client here
void Server::onReadyRead()                                          ⚠Previous definition is hereServer.cpp:153:1
{
    qDebug() << "From the on Ready Read Signal "<<Qt::endl;
    QTcpSocket* socket = qobject_cast<QTcpSocket*>(sender());
    // extract the name of sender Through is his unique socket descriptor...
    QString ClientName  = clients.key(socket);

    qDebug() << "The socket Address of the sender -> " << socket <<" His name -> " << ClientName << Qt::endl;






    // the data recived is encrypted in the client side
    QByteArray encryptedDatabyteArray = socket->readAll();
    // Decryption Key (shared with the client securely beforehand)

    QByteArray encryptionKey = "secureKey12345678"; // Same key as client
    QByteArray deCryptedData = decryptData(encryptedDatabyteArray,encryptionKey);


    QJsonDocument jsonDoc = QJsonDocument::fromJson(deCryptedData);
    QJsonObject jsonObject = jsonDoc.object();
    QString type = jsonObject["Type"].toString();
    qint32 size = jsonObject["Size"].toInt();
    QString data = jsonObject["data"].toString();
    QString command  = jsonObject["command"].toString();
    QJsonArray argsArray = jsonObject["args"].toArray();



    qDebug() << "Coming from Server Side.... " << Qt::endl;
    qDebug() << "the size of the args List sent from the user -> " << argsArray.size()<< Qt::endl;
```

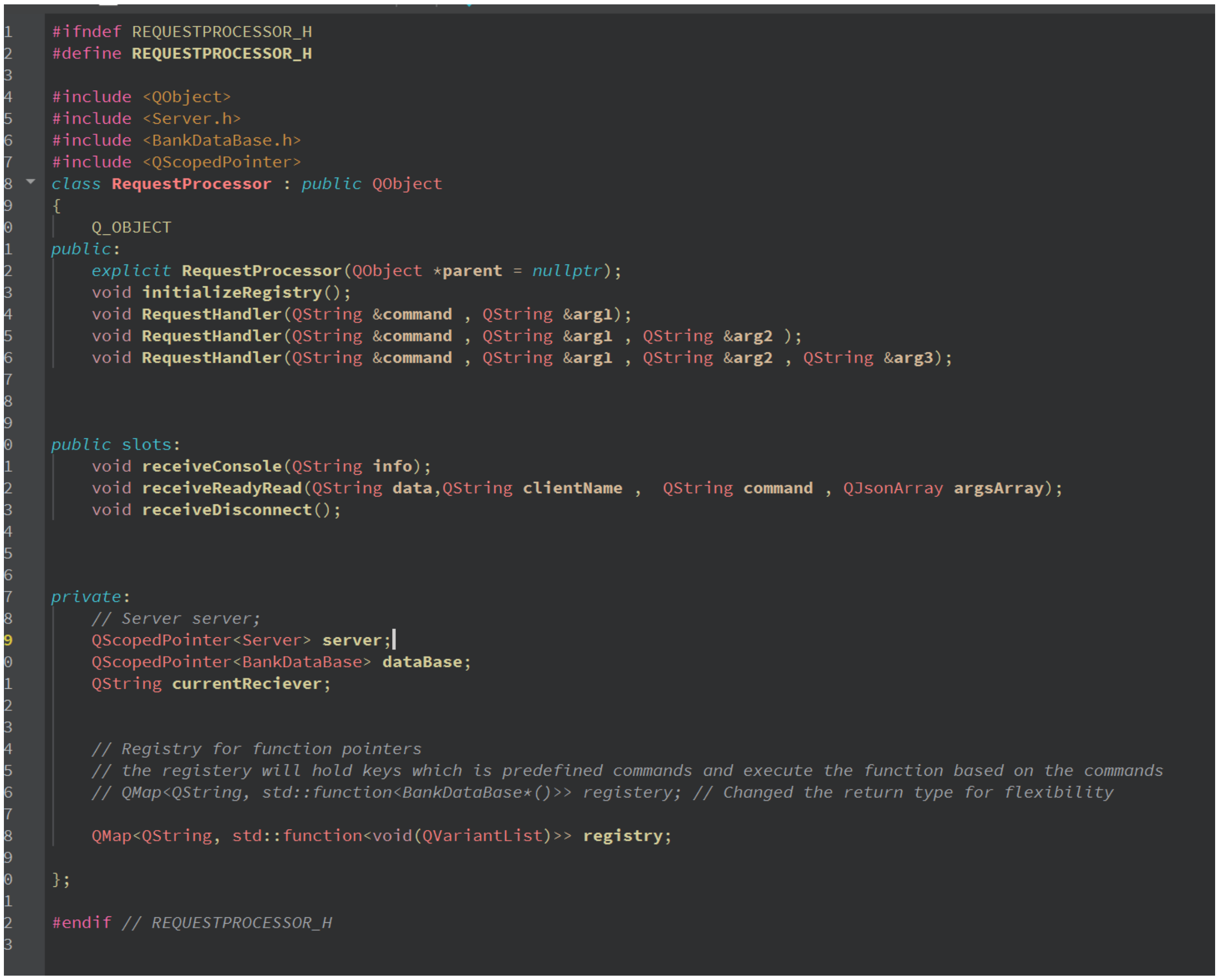
The Request Processor class is designed to efficiently handle client requests using Qt's signal-slot mechanism, a flexible request handler, and a dynamic command registry. When a client sends data, the Server class emits a signal, sendReadyRead, containing the received data, the client's name, the requested command, and the list of arguments. This signal is received in the Request Processor class at the receiveReadyRead slot.

In the receiveReadyRead slot, the server processes the incoming data by extracting essential details, including the command and its arguments. The arguments are dynamically handled and passed to a flexible RequestHandler function, which is overloaded to support different numbers of arguments. This flexibility allows the system to adapt seamlessly to commands requiring varying input parameters, ensuring smooth operation regardless of the command's complexity.

The system uses a Map-based registry to map each command string to its corresponding executable function. The registry is initialized with all the commands the moment the server runs. The registry acts as a command repository where each command is associated with a lambda expression. These lambda expressions encapsulate the logic for executing the specific command, such as login verification or transaction handling. When the RequestHandler identifies a valid command, it looks up the command in the registry, passes the required arguments as a list, and executes the corresponding function.

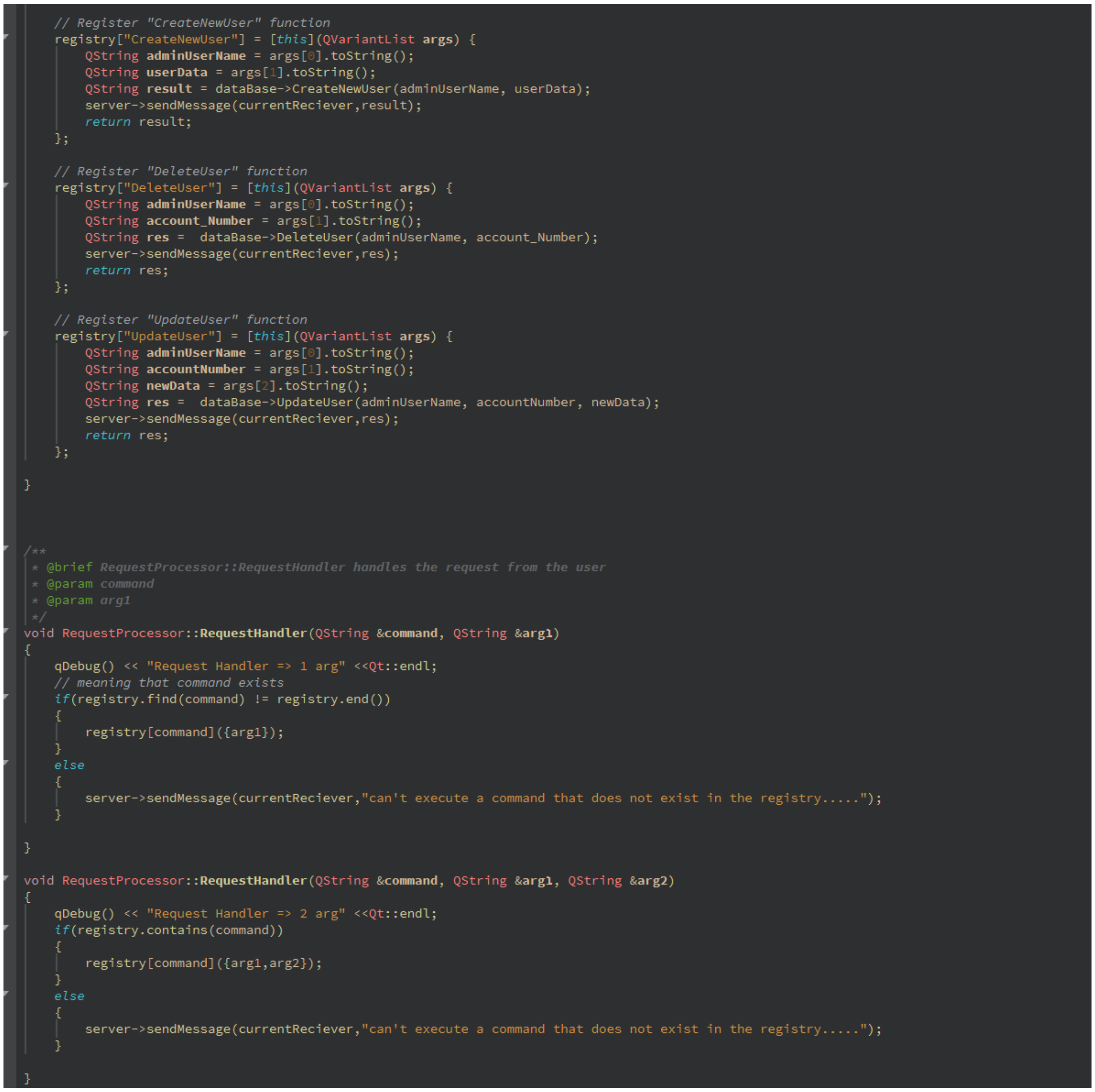
This design ensures a clean and modular approach to handling client requests. The use of signals and slots allows asynchronous communication between the server and request processor, while the registry provides a dynamic way to manage and execute commands. The combination of these components ensures that client requests are processed efficiently, commands are executed dynamically, and the system remains scalable and adaptable to new requirements.

Request processor header file



```cpp
#ifndef REQUESTPROCESSOR_H
#define REQUESTPROCESSOR_H

#include <QObject>
#include <Server.h>
#include <BankDataBase.h>
#include <QScopedPointer>
class RequestProcessor : public QObject
{
    Q_OBJECT
public:
    explicit RequestProcessor(QObject *parent = nullptr);
    void initializeRegistry();
    void RequestHandler(QString &command , QString &arg1);
    void RequestHandler(QString &command , QString &arg1 , QString &arg2 );
    void RequestHandler(QString &command , QString &arg1 , QString &arg2 , QString &arg3);



public slots:
    void receiveConsole(QString info);
    void receiveReadyRead(QString data,QString clientName ,  QString command , QJsonArray argsArray);
    void receiveDisconnect();



private:
    // Server server;
    QScopedPointer<Server> server;
    QScopedPointer<BankDataBase> dataBase;
    QString currentReciever;


    // Registry for function pointers
    // the registery will hold keys which is predefined commands and execute the function based on the commands
    // QMap<QString, std::function<BankDataBase*()>> registery; // Changed the return type for flexibility

    QMap<QString, std::function<void(QVariantList)>> registry;

};

#endif // REQUESTPROCESSOR_H
```

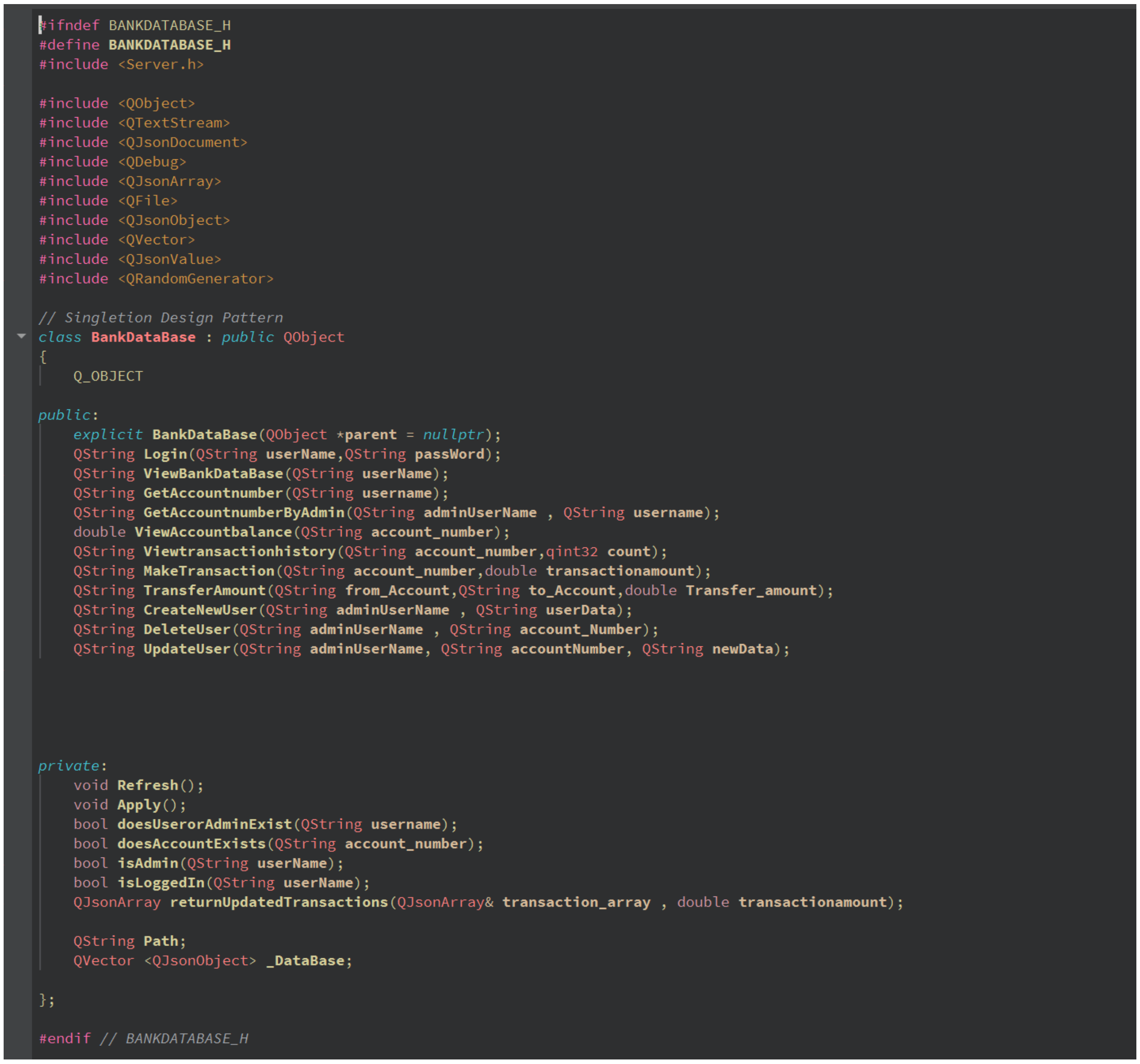
Request Processor CPP(Implementation) file



```cpp
    // Register "CreateNewUser" function
    registry["CreateNewUser"] = [this](QVariantList args) {
        QString adminUserName = args[0].toString();
        QString userData = args[1].toString();
        QString result = dataBase->CreateNewUser(adminUserName, userData);
        server->sendMessage(currentReciever,result);
        return result;
    };

    // Register "DeleteUser" function
    registry["DeleteUser"] = [this](QVariantList args) {
        QString adminUserName = args[0].toString();
        QString account_Number = args[1].toString();
        QString res =  dataBase->DeleteUser(adminUserName, account_Number);
        server->sendMessage(currentReciever,res);
        return res;
    };

    // Register "UpdateUser" function
    registry["UpdateUser"] = [this](QVariantList args) {
        QString adminUserName = args[0].toString();
        QString accountNumber = args[1].toString();
        QString newData = args[2].toString();
        QString res =  dataBase->UpdateUser(adminUserName, accountNumber, newData);
        server->sendMessage(currentReciever,res);
        return res;
    };

}



/**
 * @brief RequestProcessor::RequestHandler handles the request from the user
 * @param command
 * @param arg1
 */
void RequestProcessor::RequestHandler(QString &command, QString &arg1)
{
    qDebug() << "Request Handler => 1 arg" <<Qt::endl;
    // meaning that command exists
    if(registry.find(command) != registry.end())
    {
        registry[command]({arg1});
    }
    else
    {
        server->sendMessage(currentReciever,"can't execute a command that does not exist in the registry.....");
    }

}

void RequestProcessor::RequestHandler(QString &command, QString &arg1, QString &arg2)
{
    qDebug() << "Request Handler => 2 arg" <<Qt::endl;
    if(registry.contains(command))
    {
        registry[command]({arg1,arg2});
    }
    else
    {
        server->sendMessage(currentReciever,"can't execute a command that does not exist in the registry.....");
    }

}
```

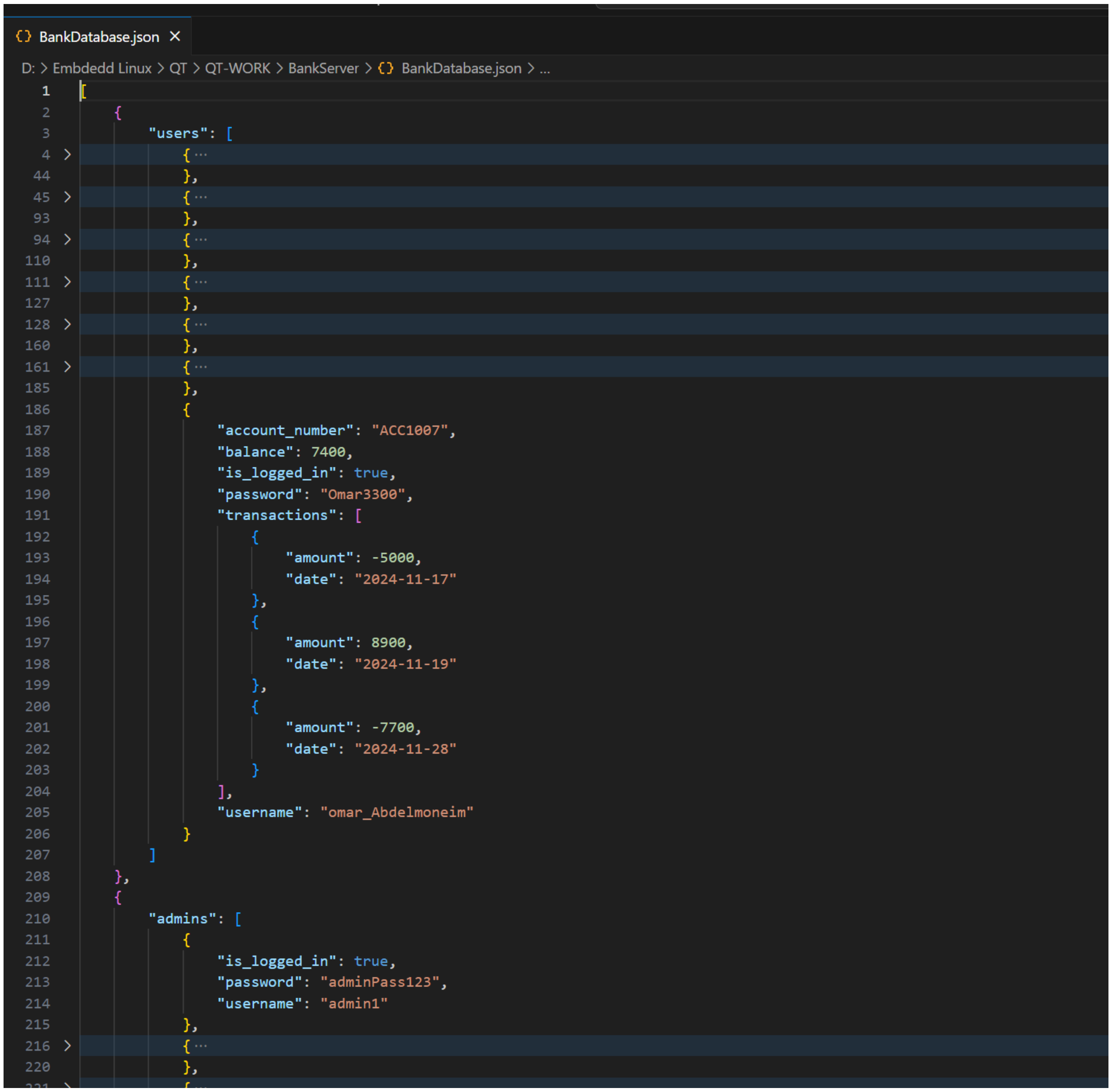
Database Class has all the logic inside where it is Implemented by using a singleton design

pattern as there is only one instance of the database class through the entire application



```cpp
#ifndef BANKDATABASE_H
#define BANKDATABASE_H
#include <Server.h>

#include <QObject>
#include <QTextStream>
#include <QJsonDocument>
#include <QDebug>
#include <QJsonArray>
#include <QFile>
#include <QJsonObject>
#include <QVector>
#include <QJsonValue>
#include <QRandomGenerator>

// Singletion Design Pattern
class BankDataBase : public QObject
{
    Q_OBJECT

public:
    explicit BankDataBase(QObject *parent = nullptr);
    QString Login(QString userName,QString passWord);
    QString ViewBankDataBase(QString userName);
    QString GetAccountnumber(QString username);
    QString GetAccountnumberByAdmin(QString adminUserName , QString username);
    double ViewAccountbalance(QString account_number);
    QString Viewtransactionhistory(QString account_number,qint32 count);
    QString MakeTransaction(QString account_number,double transactionamount);
    QString TransferAmount(QString from_Account,QString to_Account,double Transfer_amount);
    QString CreateNewUser(QString adminUserName , QString userData);
    QString DeleteUser(QString adminUserName , QString account_Number);
    QString UpdateUser(QString adminUserName, QString accountNumber, QString newData);




private:
    void Refresh();
    void Apply();
    bool doesUserorAdminExist(QString username);
    bool doesAccountExists(QString account_number);
    bool isAdmin(QString userName);
    bool isLoggedIn(QString userName);
    QJsonArray returnUpdatedTransactions(QJsonArray& transaction_array , double transactionamount);

    QString Path;
    QVector <QJsonObject> _DataBase;

};

#endif // BANKDATABASE_H
```

Bank Management Database Schema

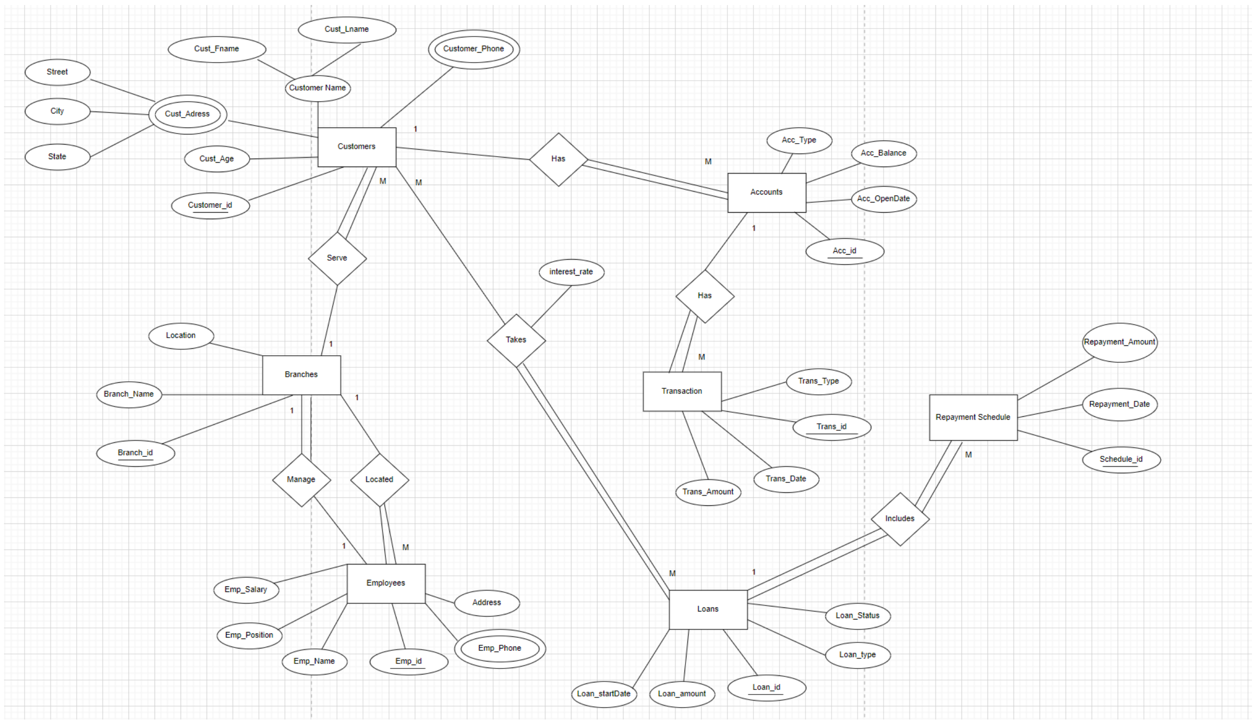


BankDatabase.json
D: > Embdedd Linux > QT > QT-WORK > BankServer > BankDatabase.json > ...

```
[
    {
        "users": [
            {···
            },
            {···
            },
            {···
            },
            {···
            },
            {···
            },
            {···
            },
            {
                "account_number": "ACC1007",
                "balance": 7400,
                "is_logged_in": true,
                "password": "Omar3300",
                "transactions": [
                    {
                        "amount": -5000,
                        "date": "2024-11-17"
                    },
                    {
                        "amount": 8900,
                        "date": "2024-11-19"
                    },
                    {
                        "amount": -7700,
                        "date": "2024-11-28"
                    }
                ],
                "username": "omar_Abdelmoneim"
            }
        ]
    },
    {
        "admins": [
            {
                "is_logged_in": true,
                "password": "adminPass123",
                "username": "admin1"
            },
            {···
            },
```

Erd Diagram of Bank DATABASE Management System

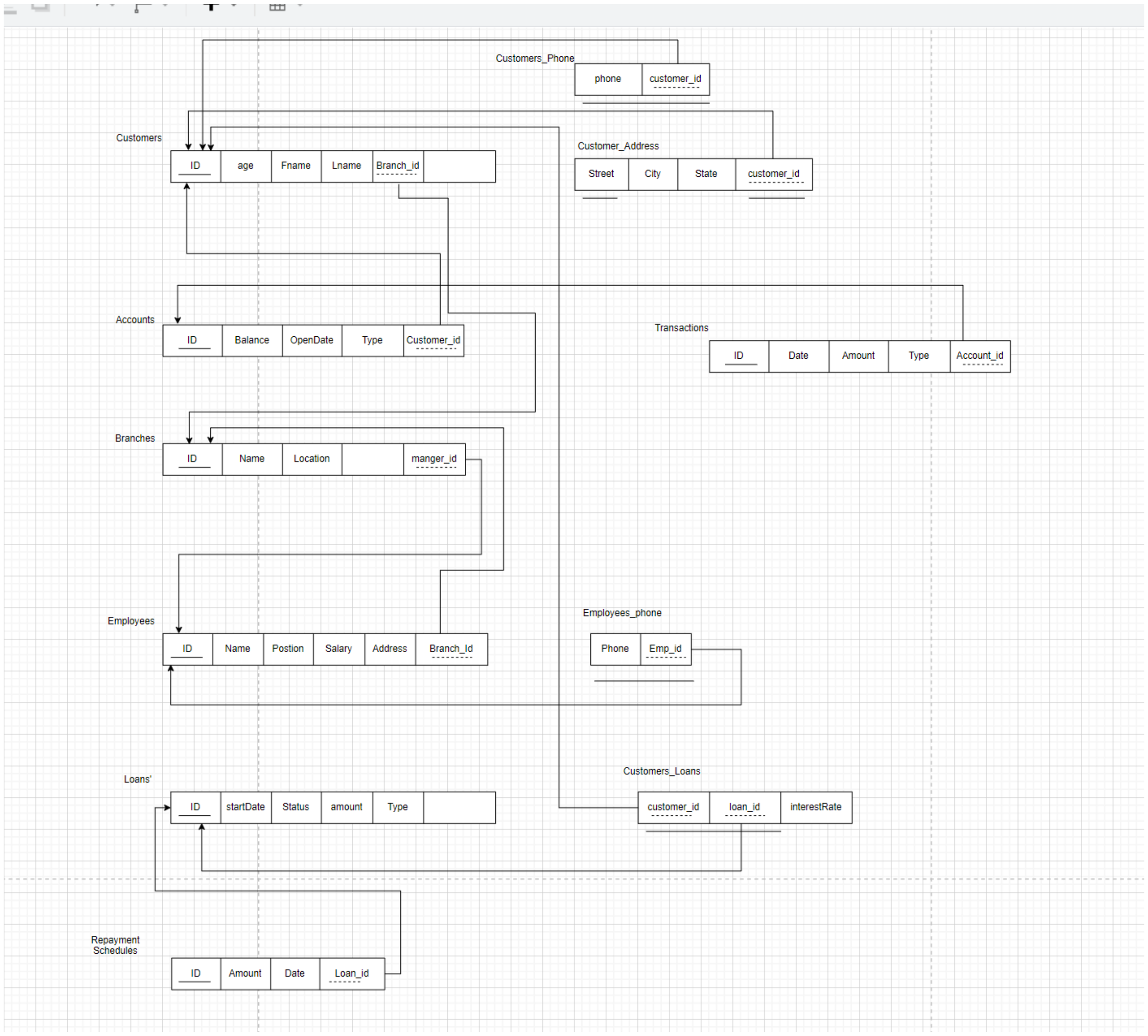
Conceptual Design:



Entity Relationships

- A **Customer** can have multiple **Accounts** and **Loans**.
- An **Account** can have multiple **Transactions**.
- A **Branch** manages multiple **Customers**, **Accounts**, **Employees**, and **Loans**.
- An **Employee** is assigned to a single **Branch** but can handle multiple **Customers** or **Loans**.
- A **Loan** is linked to a single **Customer** but involves multiple **Repayment Schedules**.

Mapping of the conceptual Design


Customers_Phone
phone
customer_id
Customers
ID
age
Fname
Lname
Branch_id
Customer_Address
Street
City
State
customer_id
Accounts
ID
Balance
OpenDate
Type
Customer_id
Transactions
ID
Date
Amount
Type
Account_id
Branches
ID
Name
Location
manger_id
Employees
ID
Name
Postion
Salary
Address
Branch_Id
Employees_phone
Phone
Emp_id
Loans'
ID
startDate
Status
amount
Type
Customers_Loans
customer_id
loan_id
interestRate
Repayment Schedules
ID
Amount
Date
Loan_id