Polytechnic of Porto
School of Engineering(ISEP)
BSc in Telecommunications and informatics Engeneering

LETI-FSOFT

TECHNICAL REPORT - 3º ITERATION

Bank Account Management System

André Moreira (1240567)
Vasco Magolo (1231562)
Francisco Silva (1230985)

Index

1.Introduction

This project focuses on the development of a banking application designed to provide users with an intuitive and secure platform for managing their financial transactions and account information. Core functionalities include account creation, user authentication, fund deposits and withdrawals, account detail viewing, and saving account balances to a file.

The application emphasizes key principles of banking systems, such as data validation, user security, and robust interaction with account features. With a focus on reliability, efficiency, and scalability, the system delivers a user-friendly experience while ensuring strong safeguards for data integrity and privacy.

2.Implementation

The Create Account Function produces a unique account number, collects the user details, and saves the new account directly into the JSON file. This ensures that each account is unique and remains immediately for future access.

```
void AccountController::createAccount() {
    int accountNumber = repository.generateUniqueAccountNumber();
    BankAccount newAccount =
view.getAccountCreationData(accountNumber);

    if (repository.saveAccount(newAccount)) {
        view.showSuccess("Account created successfully!");
        cout << "Your account number is: " << accountNumber << "\n";
        cout << "Please remember this number for future logins.\n";
    } else {
        view.showError("Failed to create account. Please try again.");
    }
}
```

Figure 1 – Function to create an account

The authentication function checks the account number and password provided against the data stored in the JSON file, ensuring that only authorized users can only reach their accounts.

```
bool AccountController::authenticateUser(int accountNumber, const
std::string &password, BankAccount &account) {
    try {
        account = repository.loadAccount(accountNumber);
        if (account.getPassword() == password) {
            return true;
        } else {
            view.showError("Invalid password.");
            return false;
        }
    } catch (const std::runtime_error &e) {
        view.showError("Account not found.");
```

```
        return false;
    }
}

bool AdminController::authenticateAdmin() {
    std::string password;
    std::cout << "Enter admin password: ";
    std::cin >> password;
    if (password == "admin123") {
        std::cout << "Authentication successful.\n";
        return true;
    } else {
        std::cout << "Authentication failed.\n";
        return false;
    }
}
```

Figure 2 – Function to authenticate User and Admin respectively

The deposit function credits the specified amount to the account balance and saves the change to the JSON file.

```
bool BankAccount::deposit(double amount) {
    if (amount <= 0) {
        cerr << "Deposit amount must be positive" << endl;
        return false;
    }
    balance += amount;
    return true;
}
```

```
void AccountController::processDeposit(BankAccount &account) {
    double amount = view.getDoubleInput("Enter deposit amount: $");
    if (amount <= 0) {
        view.showError("Deposit amount must be positive.");
        return;
    }

    if (account.deposit(amount)) {
        repository.saveAccount(account);
        view.showSuccess("Deposit successful!");
        cout << "New balance: $" << fixed << setprecision(2) <<
account.getBalance() << "\n";
    } else {
        view.showError("Deposit failed.");
    }
}
```

Figure 3 – Code to deposit funds

The withdraw function debits the specified amount from the account balance (if sufficient funds are available) and saves the updated account information to the JSON file.

```
bool BankAccount::withdraw(double amount) {
    if (amount <= 0) {
        cerr << "Withdrawal amount must be positive" << endl;
```

```
        return false;
    }
    if (amount > balance) {
        cerr << "Insufficient funds. Current balance: $" << balance <<
endl;
        return false;
    }
    balance -= amount;
    return true;
}
```

```
void AccountController::processWithdrawal(BankAccount &account) {
    double amount = view.getDoubleInput("Enter withdrawal amount: $");
    if (amount <= 0) {
        view.showError("Withdrawal amount must be positive.");
        return;
    }

    if (account.withdraw(amount)) {
        repository.saveAccount(account);
        view.showSuccess("Withdrawal successful!");
        cout << "New balance: $" << fixed << setprecision(2) <<
account.getBalance() << "\n";
    } else {
        view.showError("Withdrawal failed.");
    }
}
```

Figure 4 – Code to withdraw funds

The update account function allows users to modify their account details, such as names, address, age and passwords. After changing, the updated information is saved in the JSON file.

```
void AccountController::updateAccountInfo(BankAccount &account) {
  view.showAccountDetails(account);
  bool running = true;
  while (running) {
    view.updateAccountMenu();
    int choice = view.getIntInput("");

    switch (choice) {
      case 1:
        account.updateName(view.getNewName());
        break;
      case 2:
        account.updateAddress(view.getNewAddress());
        break;
      case 3:
        account.updateAge(view.getNewAge());
        break;
      case 4: {
        std::string newPassword = view.getNewPassword();
        if (!newPassword.empty()) {
          // The password is already hashed by getPasswordInput()
          account.updatePassword(newPassword);
        }
       break;
```

```
      }
      case 0:
        running = false;
        break;
      default:
        view.showError("Invalid choice. Please try again.");
    }
  }
  repository.saveAccount(account);
  view.showSuccess("Account information updated successfully!");
}
```

```
void AdminController::updateUserAccount() {
  int account_number;
  cout << "Enter the account number to update:\n";
  while (!(cin >> account_number)) {
    cout << "Invalid input. Please enter a valid account number:\n";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
  cin.ignore(numeric_limits<streamsize>::max(), '\n');

  try {
    BankAccount account = repository.loadAccount(account_number);
    AccountController accountController;
    accountController.updateAccountInfo(account);
    repository.saveAccount(account);
    cout << "Account information updated successfully.\n";
  } catch (const std::exception& e) {
    cout << "Account not found or error updating: " << e.what() <<
endl;
  }
}
```

Figure 5 – Code that allows the User and the Admin to update accounts respectively

The showAccountDetails function displays the account's number, name, age, address, and current balance in a clear, formatted manner on the console.

```
void ConsoleView::showAccountDetails(const BankAccount &account) const
{
  cout << "\n=== Account Details ===\n";
  cout << "Account Number: " << account.getAccountNumber() << "\n";
  cout << "Name: " << account.getName() << "\n";
  cout << "Age: " << account.getAge() << "\n";
  cout << "Address: " << account.getAddress() << "\n";
  cout << "Balance: $" << fixed << setprecision(2) <<
account.getBalance()
       << "\n";
  cout << "========================\n";
}
```

Figure 6 – Function to show account details

The generateReceipt function prints a formatted receipt to the console, which shows the account number, the name of the account holder and the current balance for the user account.

```
void AccountController::generateReceipt(const BankAccount &account) {
  cout << "\n=== Account Receipt ===\n";
  cout << "Account Number: " << account.getAccountNumber() << "\n";
  cout << "Name: " << account.getName() << "\n";
  cout << "Current Balance: $" << fixed << setprecision(2)
       << account.getBalance() << "\n";
  cout << "========================\n";
}
```

Figure 7 – Function to generate a receipt for the account

The deleteAccount function prompts the user for confirmation and, if confirmed, permanently deletes the user's account and all associated data from the system.

```
void AccountController::deleteAccount(BankAccount &account) {
  cout << "Are you sure you want to delete your account? This action
cannot be "
          "undone.\n";
  string confirmation = view.getStringInput("Type 'DELETE' to confirm:
");

  if (confirmation == "DELETE") {
    if (repository.deleteAccount(account.getAccountNumber())) {
      view.showSuccess("Account deleted successfully.");
    } else {
      view.showError("Failed to delete account.");
    }
  } else {
    view.showMessage("Account deletion cancelled.");
  }
}
```

```
void AdminController::deleteUserAccount() {
  int account_number;
  cout << "Enter the account number to delete:\n";
  while (!(cin >> account_number)) {
    cout << "Invalid input. Please enter a valid account number:\n";
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
  }
  cin.ignore(numeric_limits<streamsize>::max(), '\n');

  if (repository.deleteAccount(account_number)) {
    cout << "Account deleted successfully.\n";
  } else {
    cout << "Account not found.\n";
  }
}
```

Figure 8 – Function for the User and the Admin (respectively) to delete account

The deleteAccount function allows for the admin to have access to every account available in the JSON files

```cpp
void AdminController::viewAllAccounts() {
  ifstream file("accounts.json");
  if (!file.is_open()) {
    cerr << "Failed to open file!" << endl;
    return;
  }

  try {
    json Doc;
    file >> Doc;

    for (const auto &account : Doc) {
      cout << "Name: " << account["name"] << endl;
      cout << "Account Number: " << account["account_number"] << endl;
      cout << "Account Balance: " << account["balance"] << endl;
    }
  } catch (const json::parse_error &e) {
    cerr << "JSON parse error: " << e.what() << endl;
  }

  file.close();
}
```

Figure 9 – Function to view all the accounts

3.Unit Tests

```cpp
TEST(PasswordServiceTest, ValidPassword) {
  std::string valid = "Teste123!";
  EXPECT_TRUE(PasswordService::isValidPassword( password: valid));
}

TEST(PasswordServiceTest, InvalidPasswords) {
  EXPECT_FALSE(PasswordService::isValidPassword( password: "short"));
  EXPECT_FALSE(PasswordService::isValidPassword( password: "semDigito!"));
  EXPECT_FALSE(PasswordService::isValidPassword( password: "12345678"));
  EXPECT_FALSE(PasswordService::isValidPassword( password: "SemEspecial1"));
}

TEST(PasswordServiceTest, PasswordRequirementsMessage) {
  std::string msg = PasswordService::getPasswordRequirements();
  EXPECT_NE(msg.find( s: "least"), std::string::npos);
}
```

Figure 10 – Example of unit test that test the password requeriments

3.1Unit Tests Table

Test	Percentage	Notes
Admin – Default Constructor	100	
Admin – Parameterized Constructor	100	
Admin – Authentication Success	100	
Admin – Authentication Failure	100	
Bank Account – Deposit Increases Balance	100	
Bank Account – Withdraw Decreases Balance	100	
Bank Account – Withdraw Fails on Insufficient Funds	100	
Account Repository – Save and Load Account	100	
Account Repository – Delete Account	100	
Account Repository – Account Exists	100	
Account Repository – Generate Unique Account Number	100	
Password Service – Valid Password	100	
Password Service – Invalid Passwords	100	
Password Service – Password Requirements Message	100	
Password Service – Hash Consistency	100	
Password Service – Verify Correct Password	100	
Password Service – Verify Incorrect Password	100	
Admin Controller – Authenticate Correct Password	100	
Admin Controller – Authenticate Wrong Password	100	

Account Controller – Authenticate User Correct	100
Account Controller – Authenticate User Wrong Password	100