Solent University

**ADVANCED DATABASE SYSTEMS (QHO541)**

**Author: Enea Rina**

**Student Number: 10407625**

**Course Title: BSc (Hons) Computer sience**

**Module Lider: Dr. Muhammad Akram**

1. **Introduction to NoteBank**

The **Bank management system** called ‘NoteBank’ is a database and program solution designed to support and optimize the operations of a modern banking institution and their clients. As financial institutions handle a vast amounts of data daily, the need for a secure and efficient system has never been greater. This database system is developed to ensure management of banking processes, secure handling of sensitive data, and real-time accessibility to crucial information.

In addition to managing data, the database ensures data integrity, security, and scalability. It incorporates features like foreign key constraints, automated triggers to maintain consistency and optimize performance. Furthermore, the system's design allows it to adapt to the bank's growing needs, making it future-proof for scaling operations and handling an increasing volume of transactions and customers.

The objective of this project is to design, implement, and test a database solution that not only meets the business requirements but also provides a foundation for advanced banking features. Through normalization, query optimization, and rigorous testing, the system ensures that it can support complex banking operations while delivering an efficient and user-friendly experience through a program developed in python that lets the clients of the bank manage their accounts and money.

1. **Part 1.**
2. **Business case and requirements:**

The business case selected is a **Bank Management System** designed to manage essential and basic banking operations. The system is built to store and process data for multiple interconnected entities such as **branches**, **staff**, **users**, **accounts**, **loans**, **credit cards**, **transactions**, and **customer support**. By implementing these entities, the system provides an overview of the common bank's operations and supports key functionalities like transaction and user data and accounts processing, loan management, and customer issues.

The database is structured to ensure functionality, data integrity, and scalability. It supports features like secure data management,strong relationships between entities, and automated triggers for operational efficiency.

To establish a strong foundation for the construction and modeling of the database, seven key business requirements were taken in consideration to guide its design and implementation:

* Managing branches and their total balances while tracking details as their unique identifiers, location, number of staff and clients. For example by calculating the total assets held by each branch by summing the balances of all associated accounts.
* Tracking staff information and their role by storing and managing their data such as names, roles, email, salary, and assigned branches to easily individuate members as managers and accountants and retrieve their information if necessary.
* Recording users personal data and linking them to an account related to a specific branch. In this way we can have track of the users and the related personal account and also verify their data if necessary. All of this while implementing a cryptography algorithm to ensure the privacy of personal information's.
* Managing loans and their statuses as the system will track loans issued to users including loan amounts, interest rates and current statuses (active, paid, defaulted). In this way the system can even generate a report of overdue loans or loans with high outstanding amounts.
* Handling credit card details, the database will store credit card information, including card numbers, expiration dates, credit limits, and user associations.
* Recording all financial transactions, such as deposits, withdrawals, and transfers, will be recorded with timestamps, amount, account associations, recipient, and transaction types. In this way we can provide a statement of recent transactions for an account.
* Providing Customer support interactions that will be logged with timestamps, issues reported, and resolutions provided to ensure that the staff members can resolve the issues.

1. **Entity Relationship Diagram**

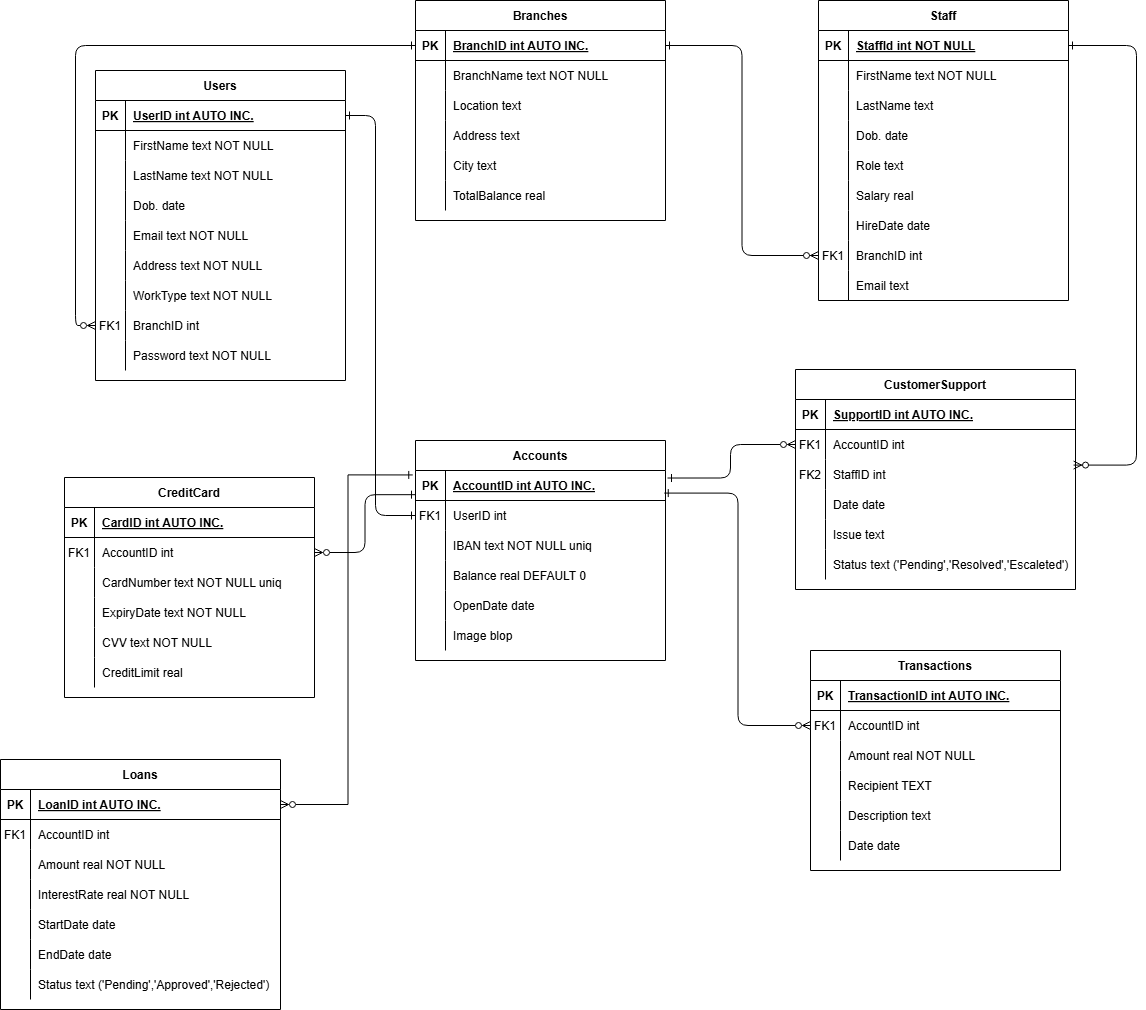
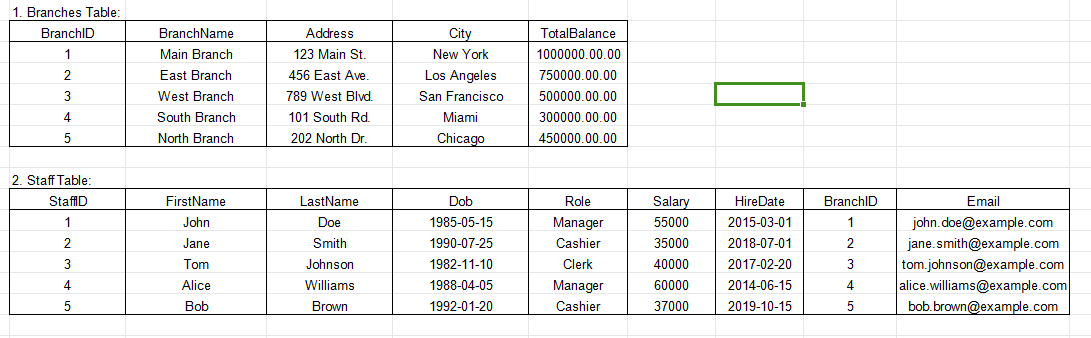
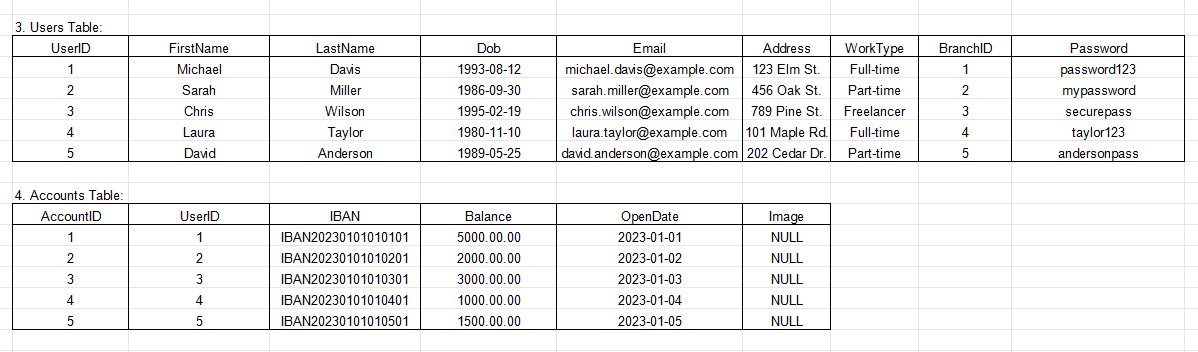
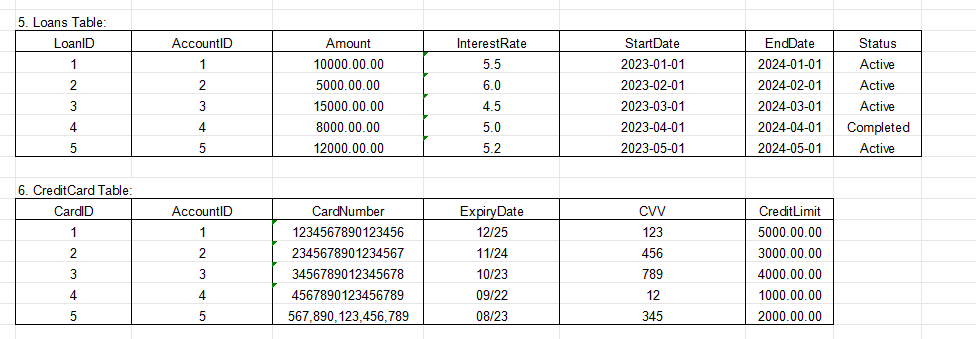
****

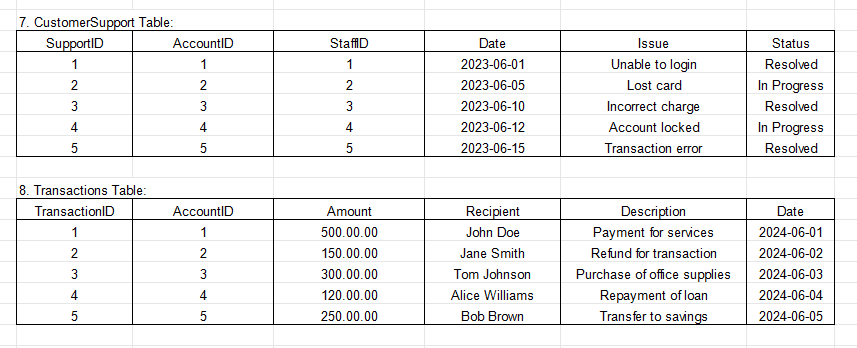
Photo 1 (Entity Relationship Database)

1. **File with collected data for the database**









1. **Sql query database development**

-- Table: Branches

CREATE TABLE Branches (

BranchID INTEGER PRIMARY KEY AUTOINCREMENT,

BranchName TEXT NOT NULL,

Address TEXT,

City TEXT,

TotalBalance REAL

);

-- Table: Staff

CREATE TABLE Staff (

StaffID INTEGER PRIMARY KEY AUTOINCREMENT,

FirstName TEXT NOT NULL,

LastName TEXT NOT NULL,

Dob DATE,

Role TEXT,

Salary REAL,

HireDate DATE,

BranchID INT,

Email TEXT,

FOREIGN KEY (BranchID) REFERENCES Branches(BranchID)

);

-- Table: Users

CREATE TABLE Users (

UserID INTEGER PRIMARY KEY AUTOINCREMENT,

FirstName TEXT NOT NULL,

LastName TEXT NOT NULL,

Dob DATE,

Email TEXT NOT NULL,

Address TEXT NOT NULL,

WorkType TEXT NOT NULL,

BranchID INT,

Password TEXT NOT NULL,

FOREIGN KEY (BranchID) REFERENCES Branches(BranchID)

);

-- Table: Accounts

CREATE TABLE Accounts (

AccountID INTEGER PRIMARY KEY AUTOINCREMENT,

UserID INT,

IBAN TEXT NOT NULL UNIQUE,

Balance REAL DEFAULT 0,

OpenDate DATE,

Image BLOB,

FOREIGN KEY (UserID) REFERENCES Users(UserID)

);

-- Table: Loans

CREATE TABLE Loans (

LoanID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

Amount REAL NOT NULL,

InterestRate REAL NOT NULL,

StartDate DATE,

EndDate DATE,

Status TEXT,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- Table: CreditCard

CREATE TABLE CreditCard (

CardID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

CardNumber TEXT NOT NULL UNIQUE,

ExpiryDate TEXT NOT NULL,

CVV TEXT NOT NULL,

CreditLimit REAL,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- Table: CustomerSupport

CREATE TABLE CustomerSupport (

SupportID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

StaffID INT,

Date DATE,

Issue TEXT,

Status TEXT,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID),

FOREIGN KEY (StaffID) REFERENCES Staff(StaffID)

);

-- Table: Transactions

CREATE TABLE Transactions (

TransactionID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

Amount REAL NOT NULL,

Recipient TEXT NOT NULL,

Description TEXT,

Date DATE,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

1. **Referential Integrity constraints**

* **Branches** is related to **Staff**, **Users**, and **Accounts** via a foreign key BranchID. This ensures that all staff, users, and accounts are correctly associated with an existing branch, allowing the system to track branch operations.
* **Staff** is connected to **CustomerSupport** via a foreign key StaffID. This ensures that every customer support ticket is handled by an existing staff member, maintaining accountability and data accuracy.
* **Users** is linked to **Accounts** via a foreign key UserID. This allows the system to track and relate all the accounts data to one user and let it log-in with his credentials.
* **Accounts** is connected to **Loans**, **CreditCard**, **Transactions**, and **CustomerSupport** via a foreign key AccountID. This ensures that all loans, credit cards, and transactions are tied to a valid account, maintaining the integrity of financial data.

1. **Part 2. Database Development**
2. **Implementation of two Triggers**

To maintain accurate branch financial data a trigger is designed to automatically updating the TotalBalance field in the Branches table whenever an account's balance is updated. It ensures that the branch's total balance reflects the sum of all associated account balances.

CREATE TRIGGER UpdateBranchBalance

AFTER UPDATE ON Accounts

FOR EACH ROW

BEGIN

UPDATE Branches

SET TotalBalance = (

SELECT COALESCE(SUM(Balance), 0)

FROM Accounts

WHERE UserID IN (

SELECT UserID

FROM Users

WHERE BranchID = (

SELECT BranchID

FROM Users

WHERE UserID = NEW.UserID

)

)

)

WHERE BranchID = (

SELECT BranchID

FROM Users

WHERE UserID = NEW.UserID

);

END;

The second trigger ensures that every new user automatically has an associated account created in the Accounts table, simplifying user onboarding and ensuring consistent user-account relationships so in this way each time a new user registers automatically an account will be created associated with that user.

CREATE TRIGGER create\_account\_after\_user\_insert

AFTER INSERT ON Users

FOR EACH ROW

BEGIN

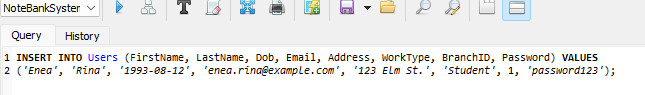
-- Insert a new account associated with the newly created user

INSERT INTO Accounts (UserID, IBAN, OpenDate)

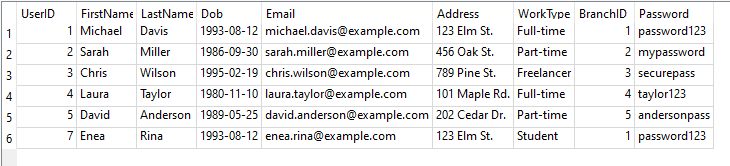
VALUES (NEW.UserID, 'IBAN' || strftime('%Y%m%d%H%M%S', 'now'), DATE('now'));

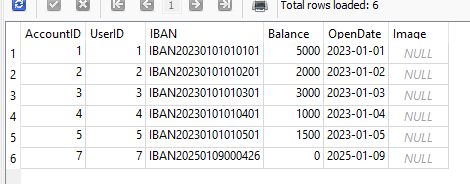
END;

Now to demonstrate the correct functionality for both the triggers a new User will be inserted to the database and automatically an account will be created in the Accounts table. After the Account is created we will set the balance of that account to £100 to demonstrate that the trigger UpdateBranchBalance will update the Main branch that is the corrispective of the User Account.

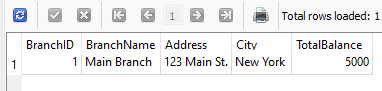


Now in the Users Table and Accounts Table in the last row we will se the new data stored



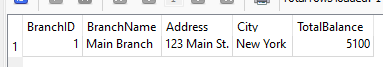


Now to test the trigger that updates automatically the Branch balance we will set the balance of the new account generated to £100 and we will see that amount added to the branch total.





Now the TotalBalance from Branches will automatically be:



1. **Implementation of two views**

A view called ActiveLoans that provides a comprehensive list of all ‘Active’ loans, including borrower details and their associated branch is implemented. This helps the business monitor active loans across branches and ensure borrowers are meeting repayment terms.

CREATE VIEW ActiveLoans AS

SELECT

L.LoanID,

L.Amount,

L.InterestRate,

L.StartDate,

L.EndDate,

L.Status,

A.IBAN,

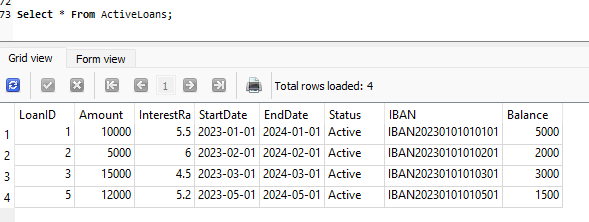
A.Balance

FROM Loans L

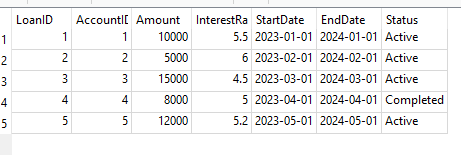
JOIN Accounts A ON L.AccountID = A.AccountID

WHERE L.Status = 'Active';

Result of the View with the data provided:



We can see that the one not Active has not been selected:



The second view called BranchPerformance summarizes each branch's financial status, current users and total stuff, providing key insights into total account balances and loan activity. This information is critical for evaluating branch performance.

CREATE VIEW BranchSummary AS

SELECT

B.BranchID,

B.BranchName,

COUNT(U.UserID) AS TotalUsers,

SUM(A.Balance) AS TotalBranchBalance,

COUNT(S.StaffID) AS TotalStaff

FROM Branches B

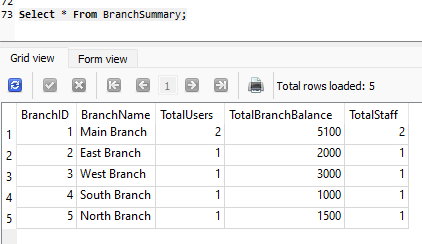
LEFT JOIN Users U ON B.BranchID = U.BranchID

LEFT JOIN Accounts A ON U.UserID = A.UserID

LEFT JOIN Staff S ON B.BranchID = S.BranchID

GROUP BY B.BranchID;

Result of the view provided:



1. **Querys that meet business requirements**
2. **Print a message:**

**To print a message in our database we create a case where we need to ensure that a transaction is not lower than 0 since it would be impossible, if the case happens a message will be printed. To meet the requirement a trigger will be created that ensures the check:**

CREATE TRIGGER check\_transaction\_amount

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

-- Ensure that the transaction amount is greater than zero

SELECT CASE

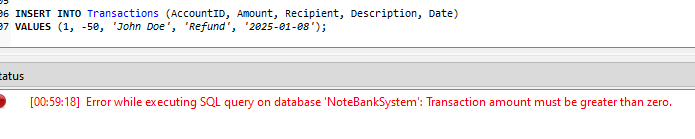
WHEN NEW.Amount <= 0 THEN

RAISE(ABORT, 'Transaction amount must be greater than zero.')

END;

END;

**Test of the query trying to insert a transaction of -50£:**



**We can also ensure to print a message with querys like:**

SELECT 'Transaction executed successfully!' AS Message, \*

FROM Transactions

WHERE TransactionID = 1;

**That provides this output:**



1. **Use of Join between two or more tables**

To demonstrate the use of JOIN between two or more tables we will recreate a first case where we need to output a list of all transactions, showing the transaction amount, date, the user who made the transaction, and the branch associated with the user.

SELECT

Transactions.TransactionID,

Transactions.Amount,

Transactions.Date,

Users.FirstName || ' ' || Users.LastName AS UserName,

Branches.BranchName AS BranchName

FROM

Transactions

JOIN

Accounts ON Transactions.AccountID = Accounts.AccountID

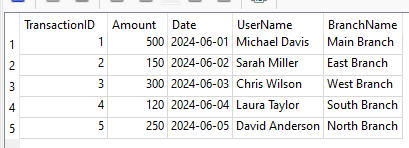
JOIN

Users ON Accounts.UserID = Users.UserID

JOIN

Branches ON Users.BranchID = Branches.BranchID;

**Output:**



**A second demonstrative query calculates the total number and amount of transactions performed at each branch.**

SELECT

Branches.BranchID,

Branches.BranchName,

COUNT(Transactions.TransactionID) AS TotalTransactions,

SUM(Transactions.Amount) AS TotalTransactionAmount

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

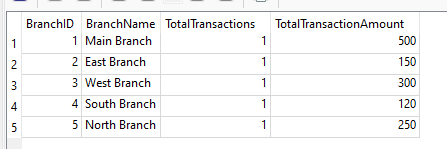
JOIN

Transactions ON Accounts.AccountID = Transactions.AccountID

GROUP BY

Branches.BranchID, Branches.BranchName

**Output:**



1. **Use of GROUP BY with HAVING**

To demonstrate a use of GROUP BY with HAVING we will be summarizing account balances by branch and filtering branches with total balances exceeding £2000.

SELECT

Branches.BranchID,

Branches.BranchName,

SUM(Accounts.Balance) AS TotalBalance

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

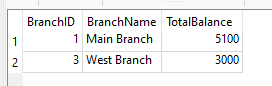
GROUP BY

Branches.BranchID, Branches.BranchName

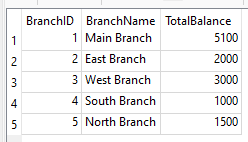
HAVING

SUM(Accounts.Balance) > 2000;

**Output:**



**If we display all the branches with their total outputs we can see that the result of our query is correct:**



**In a second case scenario we need to count the number of active loans in each branch and filters branches with more than 1 active loans.**

SELECT

Branches.BranchID,

Branches.BranchName,

COUNT(Loans.LoanID) AS ActiveLoanCount

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

JOIN

Loans ON Accounts.AccountID = Loans.AccountID

WHERE

Loans.Status = 'Active'

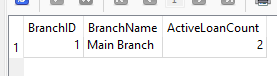
GROUP BY

Branches.BranchID, Branches.BranchName

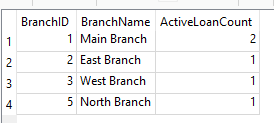
HAVING

COUNT(Loans.LoanID) > 1;

**Output:**



**If we display all of them we can see the correctness of the previous query:**

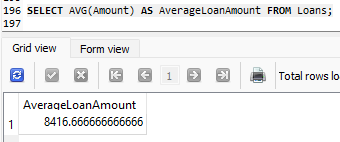


1. **Use of SQLite Functions**

In a case scenario we need to implement the AVG() function will be provided a query that calculates the **average size of loans issued** by the bank. It provides insights into the typical loan amount and helps evaluate whether the bank is meeting its loan objectives.

SELECT AVG(Amount) AS AverageLoanAmount FROM Loans;

Output:



In a second scenario to implement a function a query categorizes accounts based on their balance (LOW, MEDIUM, HIGH) , allowing the bank to **segment customers** for targeted services or promotions.

SELECT

AccountID,

IBAN,

Balance,

CASE

WHEN Balance < 1000 THEN 'Low'

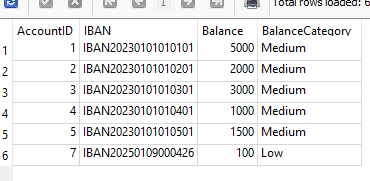
WHEN Balance BETWEEN 1000 AND 10000 THEN 'Medium'

ELSE 'High'

END AS BalanceCategory

FROM Accounts;

Output:



1. **Optimisation querys**

Query optimization is crucial for maintaining the efficiency and performance of the bank system, especially as the database grows. Without optimization, queries can become slow, consume excessive resources, and affect the user experience negatively. Optimized queries ensure faster execution, enabling the system to handle larger datasets and operations smoothly.

For example, using indexes on frequently accessed columns like AccountID or UserID improves the speed of lookups and joins by allowing the database to locate rows efficiently. Similarly, retrieving only the necessary columns instead of using SELECT \* reduces the amount of data processed, which minimizes resource usage and response times. Applying filters early in queries, such as with WHERE clauses, reduces the number of rows involved in joins or aggregations, further improving performance.

To demonstrate a correct query optimization two case scenarios will be displayed.

In the first one The following query calculates the total balance for each branch while using an index on BranchID for faster lookups.

Before query optimization:

SELECT Branches.BranchName, AVG(Loans.Amount)

FROM Branches

JOIN Users ON Branches.BranchID = Users.BranchID

JOIN Accounts ON Users.UserID = Accounts.UserID

JOIN Loans ON Accounts.AccountID = Loans.AccountID

GROUP BY Branches.BranchName;

After query optimization:

CREATE INDEX idx\_loan\_account\_id ON Loans(AccountID);

SELECT

Branches.BranchName,

AVG(Loans.Amount) AS AverageLoanAmount

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

JOIN

Loans ON Accounts.AccountID = Loans.AccountID

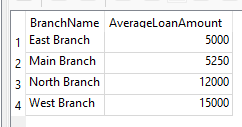
WHERE

Loans.Status = 'Active'

GROUP BY

Branches.BranchName;

Output:



In a second case we want to retrieve all transactions performed by a specific user, using indexed columns for efficient execution.

Non Optimized query:

SELECT

Transactions.TransactionID,

Transactions.Amount,

Transactions.Date,

Transactions.Description,

Users.FirstName || ' ' || Users.LastName AS UserName

FROM

Transactions

JOIN

Accounts ON Transactions.AccountID = Accounts.AccountID

JOIN

Users ON Accounts.UserID = Users.UserID

WHERE

Users.UserID = 1;

Optimized query :

CREATE INDEX idx\_user\_id ON Users(UserID);

CREATE INDEX idx\_account\_id ON Accounts(AccountID);

CREATE INDEX idx\_transaction\_account\_id ON Transactions(AccountID);

SELECT

Transactions.TransactionID,

Transactions.Amount,

Transactions.Date,

Transactions.Description,

Users.FirstName || ' ' || Users.LastName AS UserName

FROM

Transactions

JOIN

Accounts ON Transactions.AccountID = Accounts.AccountID

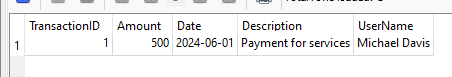
JOIN

Users ON Accounts.UserID = Users.UserID

WHERE

Users.UserID = 1;

Output:



The use of the indexes indexes ensure that the database can quickly locate relevant rows, reducing query execution time.

1. **Testing and reviewing**

Testing is important to make sure the bank database works correctly and meets the needs requirements. The testing process involves checking that tables, triggers, views, and queries perform as expected.. This process helps ensure the database is reliable and ready to be used. Some testing processes have been already covered previously as trigger and views but a further demonstration of the testing will be exampled.

The first test will be about basic query operations as insert, modify, retrieve and delete the data. To start the testing process some test the following data will be inserted to the database:

INSERT INTO Branches (BranchName, Address, City, TotalBalance)

VALUES ('Downtown', 'Liverpool St', 'Metropolis', 0);

INSERT INTO Users (FirstName, LastName, Dob, Email, Address, WorkType, BranchID, Password)

VALUES ('John', 'Doe', '1985-05-10', 'john.doe@example.com', '123 Elm St', 'Customer', 1, 'password123');

UPDATE Accounts

SET Balance = 5000 and IBAN ='IBAN20250108'

WHERE AccountID = 9;

Test 1, Inserting data through already connected tables:

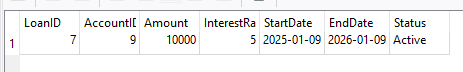
INSERT INTO Loans (AccountID, Amount, InterestRate, StartDate, EndDate, Status)

VALUES (9, 10000, 5.0, DATE('now'), DATE('now', '+1 year'), 'Active');

Test 2, Retrieve the data inserted:

SELECT \* FROM Loans WHERE AccountID = 9;

Output:



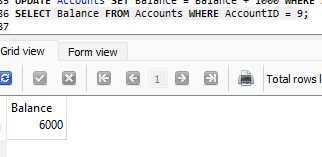
Test 3 updating existing records:

UPDATE Accounts SET Balance = Balance + 1000 WHERE AccountID = 9;

And than retriving again the data for demonstration:

SELECT Balance FROM Accounts WHERE AccountID = 9;

Output:



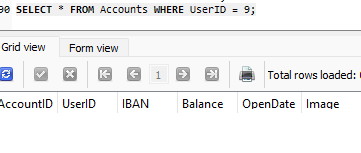
Test 4 deliting data:

DELETE FROM Users WHERE UserID = 9;

And than retrieving again to ensure the query correctness:

SELECT \* FROM Accounts WHERE UserID = 9;

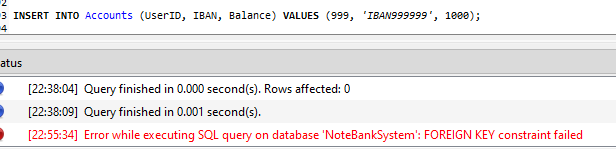
Output:



As a second case to test the database we will try to handle invalid inserted data cases as an account without a proper User connected to it:

INSERT INTO Accounts (UserID, IBAN, Balance) VALUES (999, 'IBAN999999', 1000);

Output that displays the error:



1. **Part 3 Programming for databases**
2. **Python development**

The Python Tkinter application nominated ‘NoteBank’ is designed to handle the **user side** of the bank management system. It provides a user-friendly interface for performing essential banking operations such as logging in, managing accounts, and conducting transactions. The application connects to the main SQLite database to store and retrieve data, ensuring secure and efficient data handling. Key features include user registration, password recovery, balance management, transaction tracking, and data visualization with a chart created through matplotlib. Additional functionalities like saving transaction data to an XML file and managing images linked to accounts further enhance the user experience.

Main features of the program:

· **User Login**:

* Users can log in by entering their email and password. Validation ensures secure access.

· **Account Creation**:

* New users can create an account by providing personal details such as name, email, and password.

· **Password Reset**:

* Users can reset their password if forgotten, ensuring continued access to their accounts.

· **Home Page**:

* Displays the user's account balance.
* Includes a chart showing monthly spending trends.

· **Top Up Balance**:

* Allows users to add funds to their account.

· **Perform Transactions**:

* Users can send money to another account by entering transaction details.

· **Profile Update**:

* Users can update personal details, such as name, email, or address.

· **Save or import transactions as XML**:

* Users can save their transaction history to an XML file for record-keeping.

· **Image management**:

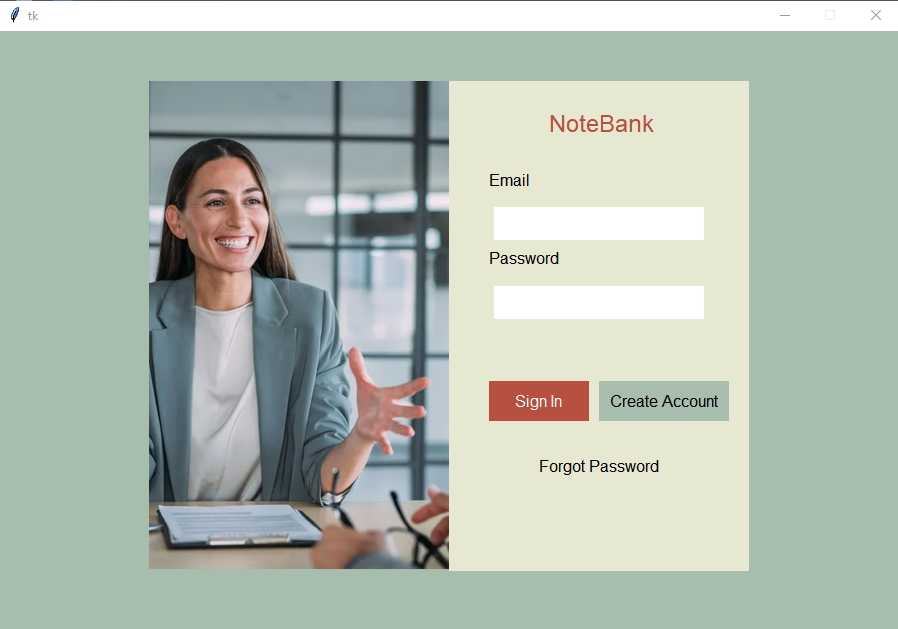
* Users can upload an image to the database linked to their account or retrieve it for viewing.

**Main structure of the Project:**

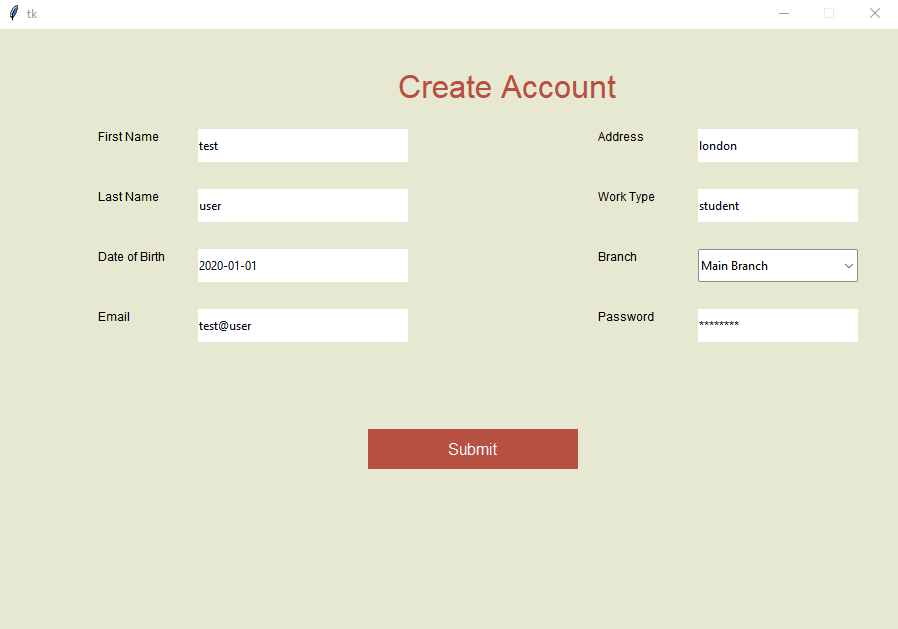
To maintain a more readable structure and order the python code have been divided int two files, The main one called ‘Main.py’ that contains the design of the GUI program and also basic functions that lets the pages communicate with each other, and than the second file called ‘Process.py’ that contains all the operational logic's and manages the communications with the database.

* **Demonstration of program execution:**

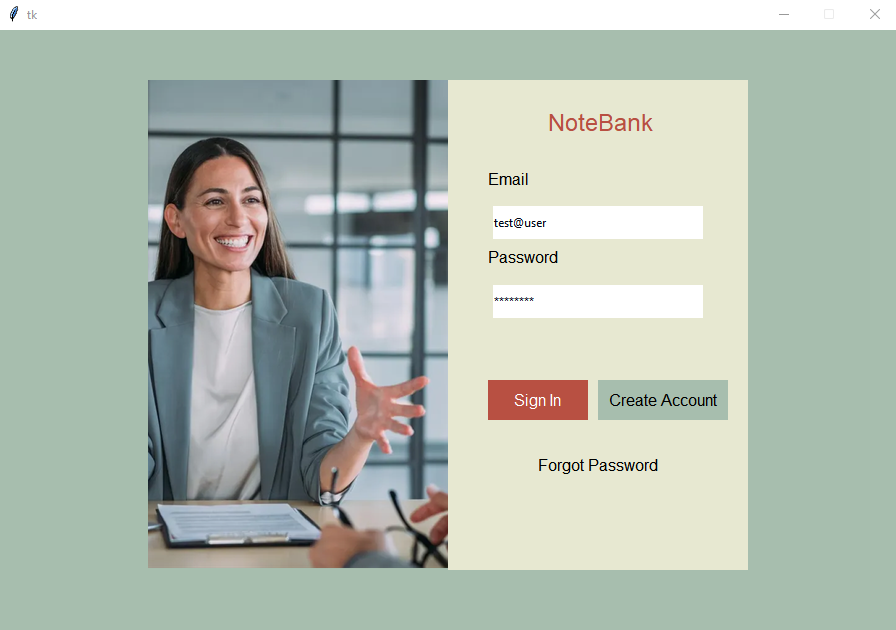
In the first execution of the program the login page is the first one to be shown:



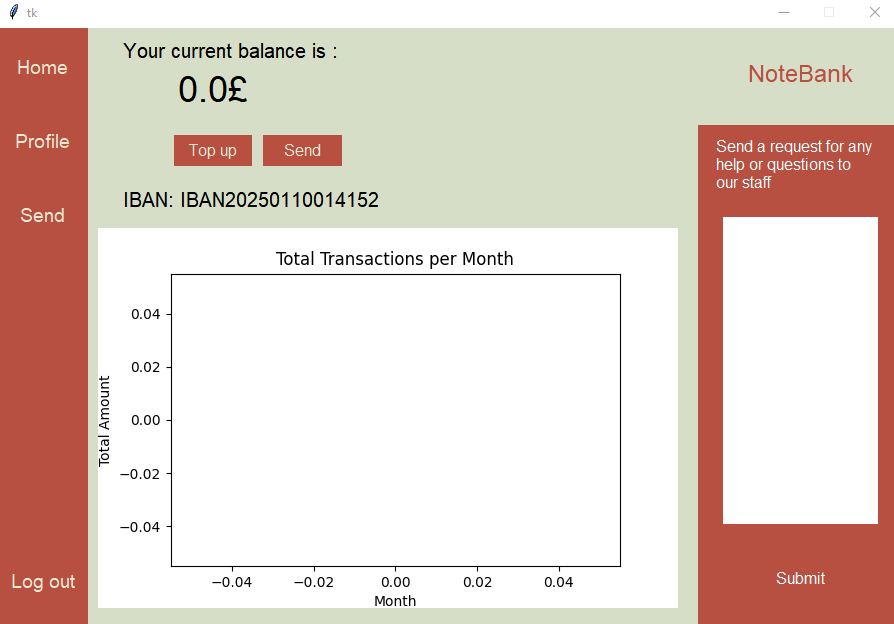
Before sign-in in we create an account clicking the button ‘Create Account’:



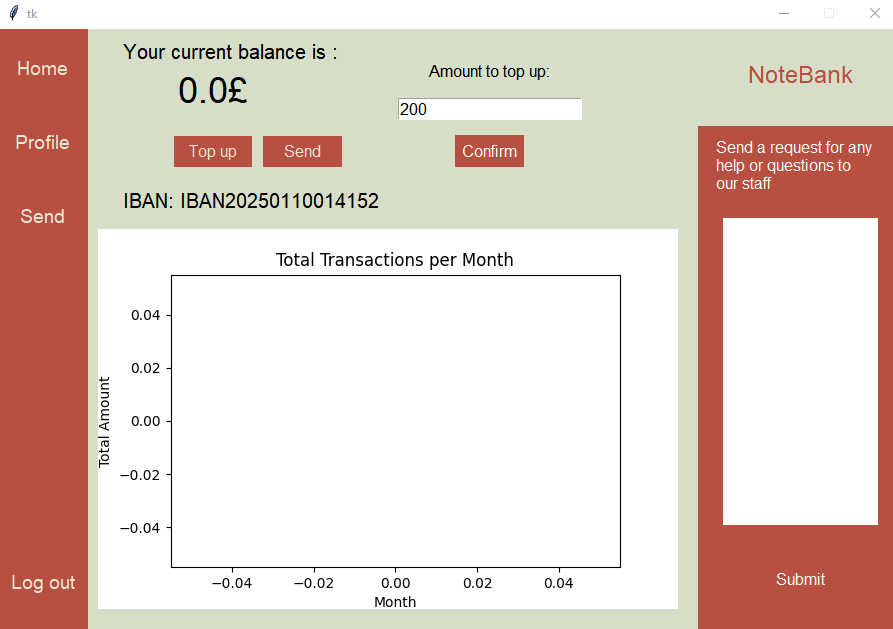
Than the program after we submit will return to the login page and here we can insert the credentials of the new account created (test@user, password):



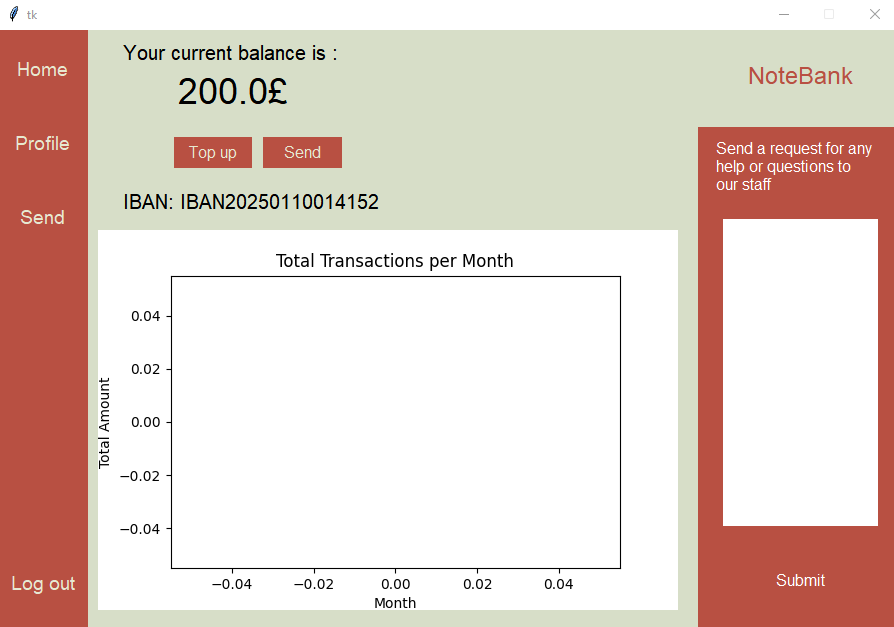
Home page shown after the login:



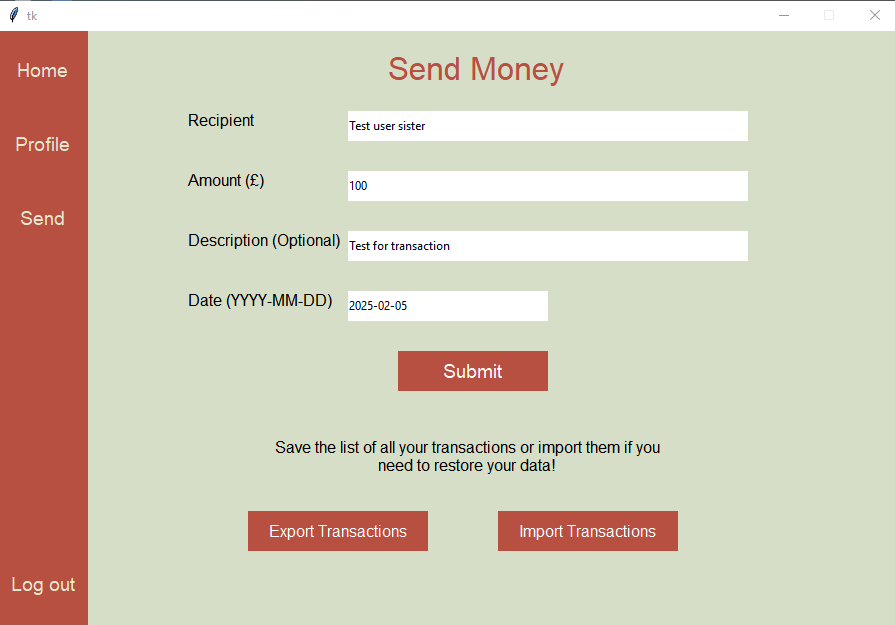
Since the account is brand new we will top up the balance by adding £200 clicking top up to open the form:



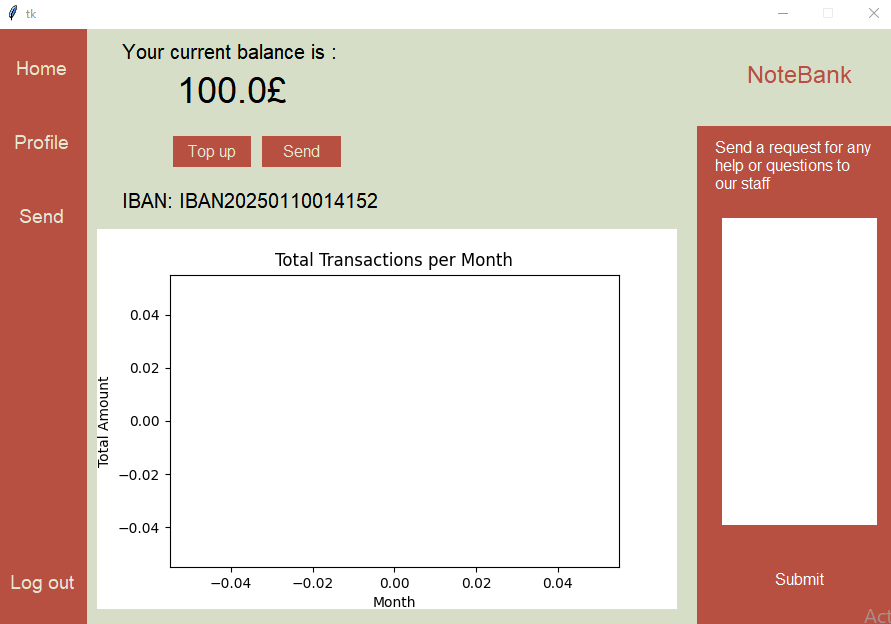
After we confirm we will see the new updated balance:



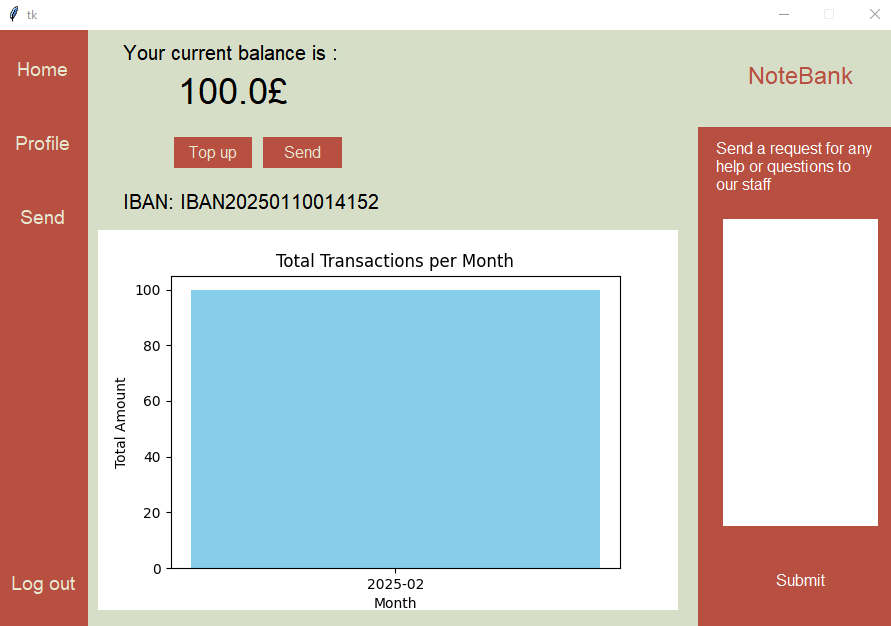
Now that we have money registered in our account we can simulate a transaction by clicking the Send button in both the left side or near top up to see the Transaction page:



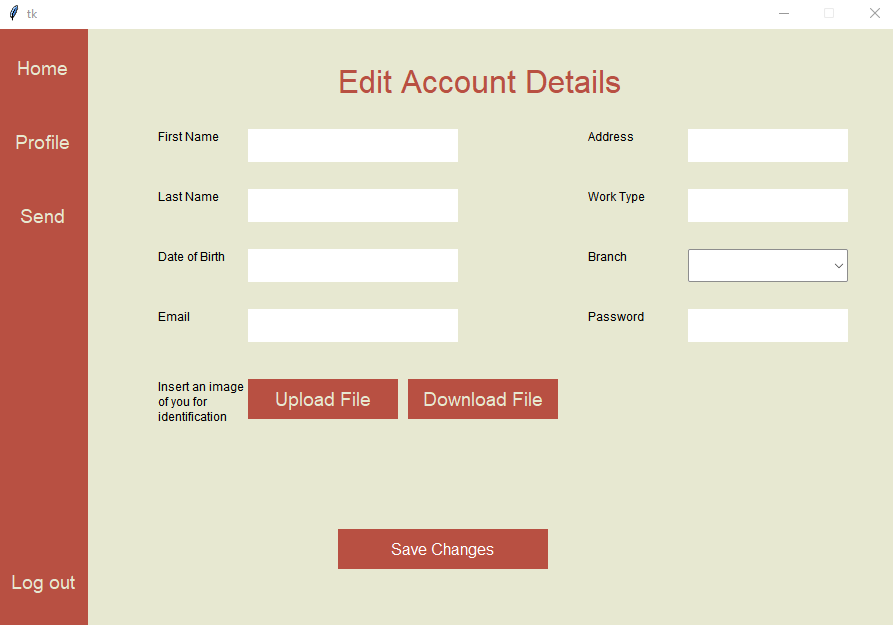
After we submit the program will return to the home page and will have the balance updated:



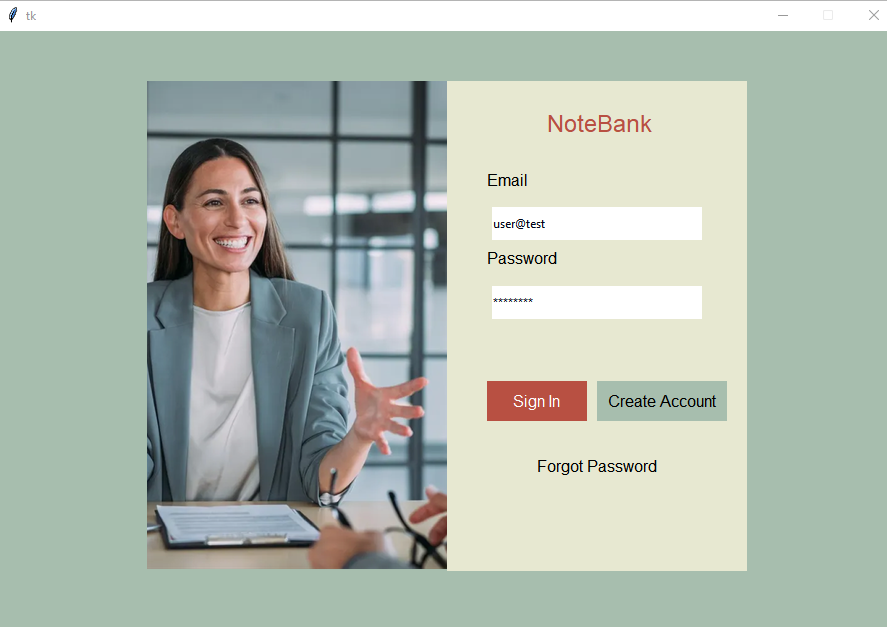
The chart in the middle of the page to display the correct amount of the transactions need that the user restarts the program and on the next execution it will display the transactions of the current year:



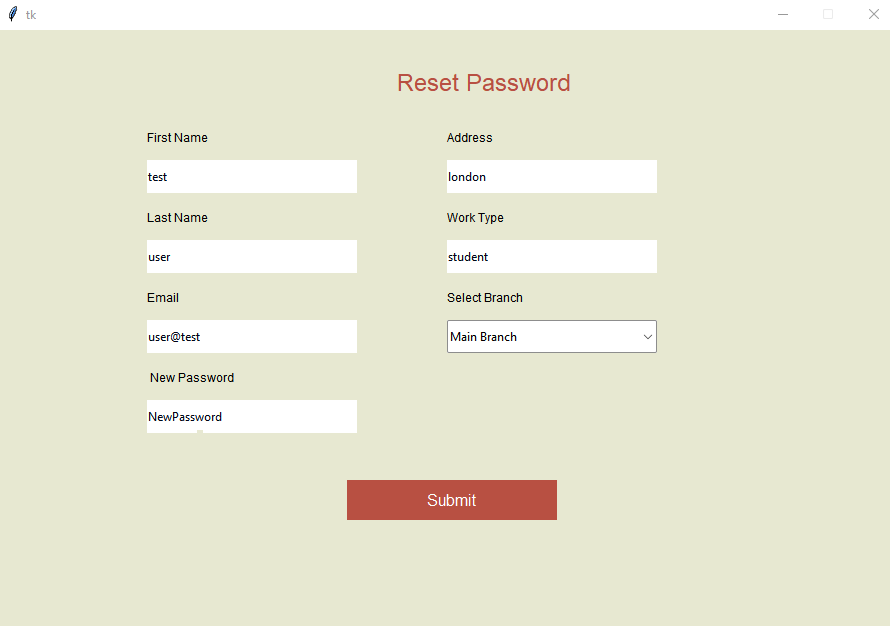
The program let’s also change the details of the profile that we inserted during the account creation by clicking on the ‘Profile’ button:



In case for example that we don’t want to change a detail we leave the field empty, to demonstrate the process we will change only the email from test@user to user@test, after we submit the program will return to the home page and there we can click the ‘Log-out’ button to log out from the account and try to log in with the new email that we provided:



Lastly, in case an user forgot the password the button ‘Forgot Password’ in the login page can be clicked and it will display a page with a form to be compiled and if the personal data inserted is correct and matches the one registered with the email than the password will be resetted:



* **Main.py code:**

from pathlib import Path

from tkinter import Tk, Canvas, Entry, Button, PhotoImage, Label, Frame, ttk, Text, filedialog

import Process

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

import matplotlib.pyplot as plt

import xml.etree.ElementTree as ET

# Define the assets path

OUTPUT\_PATH = Path(\_\_file\_\_).parent

ASSETS\_PATH = OUTPUT\_PATH / "assets" / "frame0"

def relative\_to\_assets(path: str) -> Path:

"""Returns the path to assets."""

return ASSETS\_PATH / Path(path)

# Global variable to store the current user's ID

current\_user = 0

def validation(entry1, entry2):

"""Validate user credentials and switch to the home page if correct."""

log = Process.signin(entry1, entry2)

if log[0]:

global current\_user

current\_user = log[1]

switch\_to\_home\_page()

else:

error\_label0.config(text="Invalid email or password", fg="red")

def account\_creation\_validation(firstname, lastname, dob, email, address, work, branch, password):

"""Validate and create a new account."""

log = Process.create\_account(firstname, lastname, dob, email, address, work, branch, password)

if log:

error\_label1.config(text="", fg="red")

switch\_to\_login\_page()

else:

error\_label1.config(

text="Please type correctly the data and select a valid branch\n"

"First Name, Last Name, Branch, Email, and Password are required!",

fg="red"

)

def update\_account\_validation(user, firstname, lastname, dob, email, address, work, combobox, password):

"""Validate and update account details."""

log = Process.update\_account(user, firstname, lastname, dob, email, address, work, combobox, password)

if log:

switch\_to\_home\_page()

else:

error\_label4.config(text="The branch does not exist", fg="red")

def switch\_to\_login\_page():

"""Switch back to the login page."""

error\_label.config(text="", fg="red")

error\_label0.config(text="", fg="red")

home\_frame.pack\_forget()

profile\_frame.pack\_forget()

send\_frame.pack\_forget()

create\_account\_frame.forget()

forgot\_password\_frame.forget()

entry\_1.delete(0, 'end')

entry\_2.delete(0, 'end')

login\_frame.pack(fill="both", expand=True)

login\_frame.place(relx=0.5, rely=0.5, anchor="center", width=600, height=500)

def forgot\_password\_validation(firstname, lastname, password, email, address, work, combobox):

"""Validate and reset the user's password."""

loge = Process.reset\_password(firstname, lastname, password, email, address, work, combobox)

if loge[0] and not loge[1]:

error\_label.config(text="", fg="red")

switch\_to\_login\_page()

elif not loge[0] and loge[1]:

error\_label.config(text="All fields are required.", fg="red")

elif not loge[0] and not loge[1]:

error\_label.config(text="There is no account associated with this email.", fg="red")

else:

error\_label.config(

text="Your verification credentials do not match\n the one of the account",

fg="red"

)

def handle\_upload():

"""Handle file upload by the user."""

file\_path = filedialog.askopenfilename(

title="Select a File",

filetypes=[("Image Files", "\*.jpg;\*.jpeg;\*.png;\*.gif"), ("All Files", "\*.\*")]

)

Process.upload\_file(file\_path, current\_user)

def handle\_download():

"""Handle file download by the user."""

file\_data = Process.download\_file(current\_user)

if file\_data:

save\_path = filedialog.asksaveasfilename(

title="Save File",

defaultextension=".jpg",

filetypes=[("Image Files", "\*.jpg;\*.jpeg;\*.png;\*.gif"), ("All Files", "\*.\*")]

)

if save\_path:

with open(save\_path, 'wb') as file:

file.write(file\_data)

print(f"File saved to: {save\_path}")

else:

print("Save operation canceled.")

else:

print("No file to save.")

def send\_validation(current\_user, recipient, amount, description, date):

"""Validate and process a transaction."""

log = Process.transaction(current\_user, recipient, amount, description, date)

if log == 1:

error\_label5.config(text="Fill all the required data", fg="red")

elif log == 2:

error\_label5.config(text="Error: Date must be in YYYY-MM-DD format and a valid date.", fg="red")

elif log == 3:

error\_label5.config(text="Error: Amount must be a valid number.", fg="red")

else:

switch\_to\_home\_page()

def update\_balance():

"""Fetch and display the current balance."""

current\_balance = Process.balance(current\_user)

canvas.itemconfig(balance\_text, text=f"{current\_balance[0]}£")

canvas.itemconfig(iban, text=f"IBAN: {current\_balance[1]}")

def create\_plot(transactions):

"""Create and display a plot for transactions."""

months = [transaction[0] for transaction in transactions]

amounts = [transaction[1] for transaction in transactions]

# Create a smaller plot (adjusted size)

fig, ax = plt.subplots(figsize=(5.8, 3.8))

ax.bar(months, amounts, color='skyblue')

ax.set\_xlabel("Month")

ax.set\_ylabel("Total Amount")

ax.set\_title("Total Transactions per Month")

# Embed the plot below the Iban label (adjust the position)

canvas\_plot = FigureCanvasTkAgg(fig, master=home\_frame)

canvas\_plot.draw()

canvas\_plot.get\_tk\_widget().place(x=100, y=200)

def export\_transactions\_to\_xml():

"""

Handle exporting transactions for the current user to XML.

"""

root = Process.export\_transactions(current\_user)

# Prompt the user to save the file

save\_path = filedialog.asksaveasfilename(

title="Save Transactions as XML",

defaultextension=".xml",

filetypes=[("XML Files", "\*.xml")]

)

if save\_path:

tree = ET.ElementTree(root)

tree.write(save\_path, encoding="utf-8", xml\_declaration=True)

print(f"Transactions exported to {save\_path}")

def import\_transactions\_from\_xml():

"""

Handle importing transactions from an XML file.

"""

# Prompt the user to open a file

file\_path = filedialog.askopenfilename(

title="Open Transactions XML File",

filetypes=[("XML Files", "\*.xml")]

)

if file\_path:

success = Process.import\_transactions\_from\_xml(file\_path)

if success:

print("Transactions imported successfully.")

else:

print("Failed to import transactions.")

def switch\_to\_home\_page():

"""Switch to the home page."""

profile\_frame.pack\_forget()

send\_frame.pack\_forget()

update\_balance()

login\_frame.pack\_forget()

create\_account\_frame.forget()

branch\_combobo.set("")

home\_frame.pack(fill="both", expand=True)

def switch\_to\_create\_account\_page():

"""Switch to the create account page."""

login\_frame.pack\_forget()

home\_frame.pack\_forget()

entry\_firstnam.delete(0, 'end')

entry\_lastnam.delete(0, 'end')

entry\_do.delete(0, 'end')

entry\_emai.delete(0, 'end')

entry\_addres.delete(0, 'end')

entry\_worktyp.delete(0, 'end')

entry\_passwor.delete(0, 'end')

create\_account\_frame.pack(fill="both", expand=True)

def switch\_to\_forgot\_password():

"""Clear the create account fields before switching to forgot password page."""

login\_frame.pack\_forget()

forgot\_password\_frame.pack(fill="both", expand=True)

def switch\_to\_profile\_page():

"""Switch to the profile page."""

home\_frame.pack\_forget()

send\_frame.pack\_forget()

entry\_firstnam.delete(0, 'end')

entry\_lastnam.delete(0, 'end')

entry\_do.delete(0, 'end')

entry\_emai.delete(0, 'end')

entry\_addres.delete(0, 'end')

entry\_worktyp.delete(0, 'end')

entry\_passwor.delete(0, 'end')

profile\_frame.pack(fill="both", expand=True)

def switch\_to\_send\_page():

"""Switch to the send money page."""

home\_frame.pack\_forget()

profile\_frame.pack\_forget()

entry\_recipient.delete(0, 'end')

entry\_amount.delete(0, 'end')

entry\_description.delete(0, 'end')

entry\_date.delete(0, 'end')

send\_frame.pack(fill="both", expand=True)

def show\_top\_up\_form():

"""Create the top-up form for adding funds."""

top\_up\_frame = Frame(home\_frame, bg="#D7DEC8", width=300, height=150)

top\_up\_frame.place(x=400, y=20) # Position the form on the screen

# Label for amount to top up

amount\_label = Label(top\_up\_frame, text="Amount to top up:", bg="#D7DEC8", font=("Inter", 12))

amount\_label.pack(pady=10)

# Entry field to input the top-up amount

quantity = Entry(top\_up\_frame, font=("Inter", 12))

quantity.pack(pady=5)

# Function to handle the confirm button click

def confirm\_top\_up():

amount = quantity.get()

if amount.isdigit() and float(amount) > 0:

Process.top\_up(amount, current\_user)

top\_up\_frame.destroy() # Close the top-up form

switch\_to\_home\_page()

# Confirm button

confirm\_button = Button(

top\_up\_frame,

text="Confirm",

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat",

command=confirm\_top\_up

)

confirm\_button.pack(pady=10)

# Main window

window = Tk()

window.geometry("900x600")

window.configure(bg="#A7BEAE")

#---------------------------------------------------

# Login Frame

login\_frame = Frame(window, bg="#A7BEAE")

login\_frame.pack(fill="both", expand=True)

login\_frame.place(relx=0.5, rely=0.5, anchor="center", width=600, height=500) # Centered within the 900x600 window

canvas = Canvas(

login\_frame,

bg="#A7BEAE",

height=500,

width=600,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas.place(x=0, y=-10)

canvas.create\_rectangle(300.0, 0.0, 600.0, 500.0, fill="#E7E8D1", outline="")

canvas.create\_text(400.0, 40.0, anchor="nw", text="NoteBank", fill="#B85042", font=("Inter SemiBold", 24 \* -1))

image\_image\_1 = PhotoImage(file=relative\_to\_assets("image\_1.png"))

canvas.create\_image(150.0, 250.0, image=image\_image\_1)

canvas.create\_text(340.0, 100.0, anchor="nw", text="Email", fill="#000000", font=("Inter", 16 \* -1))

entry\_1 = Entry(login\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_1.place(x=345.0, y=126.0, width=210.0, height=33.0)

canvas.create\_text(340.0, 178.0, anchor="nw", text="Password", fill="#000000", font=("Inter", 16 \* -1))

entry\_2 = Entry(login\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0, show="\*")

entry\_2.place(x=345.0, y=205.0, width=210.0, height=33.0)

error\_label0 = Label(login\_frame, text="", bg="#E7E8D1", fg="red", font=("Inter", 10))

error\_label0.place(x=345, y=250)

button\_1 = Button(

login\_frame,

text="Sign In",

command=lambda: validation(entry\_1, entry\_2),

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

button\_1.place(x=340.0, y=300.0, width=100.0, height=40.0)

button\_2 = Button(

login\_frame,

text="Create Account",

command=lambda: switch\_to\_create\_account\_page(),

bg="#A7BEAE",

fg="#000000",

font=("Inter", 12),

relief="flat"

)

button\_2.place(x=450.0, y=300.0, width=130.0, height=40.0)

button\_3 = Button(

login\_frame,

text="Forgot Password",

command=lambda: switch\_to\_forgot\_password(),

bg="#E7E8D1",

fg="#000000",

font=("Inter", 12),

relief="flat"

)

button\_3.place(x=360.0, y=370.0, width=180.0, height=30.0)

#-------------------------------------------------------------------------

home\_frame = Frame(window, bg="#D7DEC8")

home\_frame.pack(fill="both", expand=True)

home\_frame.pack\_forget()

canvas = Canvas(

home\_frame,

bg="#D7DEC8",

height=600,

width=900,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas.place(x=0, y=0)

# "Your current balance is" label

canvas.create\_text(

125.0,

11.0,

anchor="nw",

text="Your current balance is :",

fill="#000000",

font=("Inter", 20 \* -1)

)

# Display balance

balance\_text = canvas.create\_text(

180.0,

41.0,

anchor="nw",

text="Loading error",

fill="#000000",

font=("Inter", 36 \* -1)

)

iban = canvas.create\_text(

125.0,

160.0,

anchor="nw",

text="Iban",

fill="#000000",

font=("Inter", 20 \* -1)

)

# "Top up" and "Send" buttons

canvas.create\_rectangle(

176.0,

107.0,

254.0,

138.0,

fill="#B85042",

outline=""

)

canvas.create\_rectangle(

265.0,

107.0,

344.0,

138.0,

fill="#B85042",

outline=""

)

# "Top up" and "Send" buttons

Button(

home\_frame,

text="Top up",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 12),

relief="flat",

command=show\_top\_up\_form # Show the top-up form when clicked

).place(x=176.0, y=107.0, width=78.0, height=31.0)

Button(

home\_frame,

text="Send",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 12),

relief="flat",

command=lambda: switch\_to\_send\_page()

).place(x=265.0, y=107.0, width=79.0, height=31.0)

# Left Sidebar - Home, Profile, Cards, Loans, etc.

sidebar\_frame = Frame(home\_frame, bg="#B85042", width=90, height=600)

sidebar\_frame.place(x=0, y=0)

Button(

sidebar\_frame,

text="Home",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:switch\_to\_home\_page()

).place(x=0, y=19, width=90, height=40)

Button(

sidebar\_frame,

text="Profile",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:switch\_to\_profile\_page()

).place(x=0, y=93, width=90, height=40)

Button(

sidebar\_frame,

text="Send",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:switch\_to\_send\_page()

).place(x=0, y=167, width=90, height=40)

Button(

sidebar\_frame,

text="Log out",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:switch\_to\_login\_page()

).place(x=0, y=533, width=90, height=40)

# "NoteBank" text at the top

canvas.create\_text(

750.0,

33.0,

anchor="nw",

text="NoteBank",

fill="#B85042",

font=("Inter SemiBold", 24 \* -1)

)

transaction = Process.list\_transactions()

# Call the function to display the table and plot

create\_plot(transaction)

# Right Side - Help Text Area

canvas.create\_rectangle(

700.0,

97.0,

900.0,

600.0,

fill="#B85042",

outline=""

)

canvas.create\_text(

718.0,

109.0,

anchor="nw",

text="Send a request for any\nhelp or questions to\nour staff",

fill="#FFFFFF",

font=("Inter SemiBold", 16 \* -1)

)

# Textbox for request input

request\_input = Text(

home\_frame,

bd=0,

bg="#FFFFFF",

fg="#000716",

wrap="word", # Automatically wrap text at word boundaries

font=("Inter", 12) # Adjust font size if needed

)

request\_input.place(x=725.0, y=189.0, width=155.0, height=307.0)

request\_input.insert("1.0", "") # Start text from the top-left

# Submit Button for request

submit\_button = Button(

home\_frame,

text="Submit",

command=lambda: switch\_to\_home\_page(),

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

submit\_button.place(x=725.0, y=530.0, width=155.0, height=40.0)

#Now there will be the frame of the create account

#------------------------------------------------------------------------------

create\_account\_frame = Frame(window, bg="#E7E8D1")

canvas\_create = Canvas(

create\_account\_frame,

bg="#E7E8D1",

height=600,

width=900,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas\_create.place(x=0, y=0)

canvas\_create.create\_text(400.0, 40.0, anchor="nw", text="Create Account", fill="#B85042", font=("Inter SemiBold", 24))

# Left Column Inputs

canvas\_create.create\_text(100.0, 100.0, anchor="nw", text="First Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_firstn = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_firstn.place(x=200.0, y=100.0, width=210.0, height=33.0)

canvas\_create.create\_text(100.0, 160.0, anchor="nw", text="Last Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_lastn = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_lastn.place(x=200.0, y=160.0, width=210.0, height=33.0)

canvas\_create.create\_text(100.0, 220.0, anchor="nw", text="Date of Birth", fill="#000000", font=("Inter", 12 \* -1))

entry\_d = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_d.place(x=200.0, y=220.0, width=210.0, height=33.0)

canvas\_create.create\_text(100.0, 280.0, anchor="nw", text="Email", fill="#000000", font=("Inter", 12 \* -1))

entry\_em = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_em.place(x=200.0, y=280.0, width=210.0, height=33.0)

# Right Column Inputs (add some space to avoid being too close to the border)

canvas\_create.create\_text(600.0, 100.0, anchor="nw", text="Address", fill="#000000", font=("Inter", 12 \* -1))

entry\_addr = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_addr.place(x=700.0, y=100.0, width=160.0, height=33.0)

canvas\_create.create\_text(600.0, 160.0, anchor="nw", text="Work Type", fill="#000000", font=("Inter", 12 \* -1))

entry\_workt = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_workt.place(x=700.0, y=160.0, width=160.0, height=33.0)

canvas\_create.create\_text(600.0, 220.0, anchor="nw", text="Branch", fill="#000000", font=("Inter", 12 \* -1))

# Load the branches into the combobox

branches = Process.load\_branches()

branch\_comb = ttk.Combobox(create\_account\_frame, values=branches)

branch\_comb.place(x=700.0, y=220.0, width=160.0, height=33.0)

canvas\_create.create\_text(600.0, 280.0, anchor="nw", text="Password", fill="#000000", font=("Inter", 12 \* -1))

entry\_passw = Entry(create\_account\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0, show="\*")

entry\_passw.place(x=700.0, y=280.0, width=160.0, height=33.0)

error\_label1 = Label(create\_account\_frame, text="", bg="#E7E8D1", fg="red", font=("Inter", 10))

error\_label1.place(x=250, y=460)

submit\_button = Button(

create\_account\_frame,

text="Submit",

command=lambda: account\_creation\_validation(entry\_firstn, entry\_lastn, entry\_d, entry\_em, entry\_addr, entry\_workt, branch\_comb, entry\_passw),

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

submit\_button.place(x=370.0, y=400.0, width=210.0, height=40.0)

#---------------------------------------------------------------------------------------------

forgot\_password\_frame = Frame(window, bg="#E7E8D1")

forgot\_password\_frame.pack\_forget()

canvas\_forgot\_password = Canvas(

forgot\_password\_frame,

bg="#E7E8D1",

height=600,

width=900,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas\_forgot\_password.place(x=0, y=0)

canvas\_forgot\_password.create\_rectangle(0.0, 0.0, 900.0, 600.0, fill="#E7E8D1", outline="")

canvas\_forgot\_password.create\_text(400.0, 40.0, anchor="nw", text="Reset Password", fill="#B85042", font=("Inter SemiBold", 24 \* -1))

# First Name

canvas\_forgot\_password.create\_text(150.0, 100.0, anchor="nw", text="First Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_firstname = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_firstname.place(x=150.0, y=130.0, width=210.0, height=33.0)

# Last Name

canvas\_forgot\_password.create\_text(150.0, 180.0, anchor="nw", text="Last Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_lastname = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_lastname.place(x=150.0, y=210.0, width=210.0, height=33.0)

# Email

canvas\_forgot\_password.create\_text(150.0, 260.0, anchor="nw", text="Email", fill="#000000", font=("Inter", 12 \* -1))

entry\_email = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_email.place(x=150.0, y=290.0, width=210.0, height=33.0)

#Password

canvas\_forgot\_password.create\_text(150.0, 340.0, anchor="nw", text=" New Password", fill="#000000", font=("Inter", 12 \* -1))

entry\_password = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_password.place(x=150.0, y=370.0, width=210.0, height=33.0)

# Address

canvas\_forgot\_password.create\_text(450.0, 100.0, anchor="nw", text="Address", fill="#000000", font=("Inter", 12 \* -1))

entry\_address = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_address.place(x=450.0, y=130.0, width=210.0, height=33.0)

# Work Type

canvas\_forgot\_password.create\_text(450.0, 180.0, anchor="nw", text="Work Type", fill="#000000", font=("Inter", 12 \* -1))

entry\_worktype = Entry(forgot\_password\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_worktype.place(x=450.0, y=210.0, width=210.0, height=33.0)

# Branch

canvas\_forgot\_password.create\_text(450.0, 260.0, anchor="nw", text="Select Branch", fill="#000000", font=("Inter", 12 \* -1))

branches = Process.load\_branches()

branch\_combobox = ttk.Combobox(forgot\_password\_frame, values=branches)

branch\_combobox.place(x=450.0, y=290.0, width=210.0, height=33.0)

# Error Label

error\_label = Label(forgot\_password\_frame, text="", bg="#E7E8D1", fg="red", font=("Inter", 10))

error\_label.place(x=200.0, y=400.0)

# Submit Button

submit\_button = Button(

forgot\_password\_frame,

text="Submit",

command=lambda: forgot\_password\_validation(entry\_firstname, entry\_lastname, entry\_password, entry\_email, entry\_address,

entry\_worktype, branch\_combobox),

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

submit\_button.place(x=350.0, y=450.0, width=210.0, height=40.0)

#--------------------------------------------

# Profile Frame

profile\_frame = Frame(window, bg="#D7DEC8")

profile\_frame.pack(fill="both", expand=True)

profile\_frame.pack\_forget()

# Sidebar (Left Side)

sidebar\_frame\_profile = Frame(profile\_frame, bg="#B85042", width=90, height=600)

sidebar\_frame\_profile.place(x=0, y=0)

Button(

sidebar\_frame\_profile,

text="Home",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_home\_page()

).place(x=0, y=19, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Profile",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_profile\_page()

).place(x=0, y=93, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Send",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_send\_page()

).place(x=0, y=167, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Log out",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_login\_page()

).place(x=0, y=533, width=90, height=40)

# Form Section (Right Side)

form\_frame = Frame(profile\_frame, bg="#E7E8D1", width=810, height=600)

form\_frame.place(x=90, y=0)

canvas\_form = Canvas(

form\_frame,

bg="#E7E8D1",

height=600,

width=810,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas\_form.place(x=0, y=0)

canvas\_form.create\_text(250.0, 35.0, anchor="nw", text="Edit Account Details", fill="#B85042", font=("Inter SemiBold", 24))

# Left Column Inputs

canvas\_form.create\_text(70.0, 100.0, anchor="nw", text="First Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_firstnam = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_firstnam.place(x=160.0, y=100.0, width=210.0, height=33.0)

canvas\_form.create\_text(70.0, 160.0, anchor="nw", text="Last Name", fill="#000000", font=("Inter", 12 \* -1))

entry\_lastnam = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_lastnam.place(x=160.0, y=160.0, width=210.0, height=33.0)

canvas\_form.create\_text(70.0, 220.0, anchor="nw", text="Date of Birth", fill="#000000", font=("Inter", 12 \* -1))

entry\_do = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_do.place(x=160.0, y=220.0, width=210.0, height=33.0)

canvas\_form.create\_text(70.0, 280.0, anchor="nw", text="Email", fill="#000000", font=("Inter", 12 \* -1))

entry\_emai = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_emai.place(x=160.0, y=280.0, width=210.0, height=33.0)

# Right Column Inputs

canvas\_form.create\_text(500.0, 100.0, anchor="nw", text="Address", fill="#000000", font=("Inter", 12 \* -1))

entry\_addres = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_addres.place(x=600.0, y=100.0, width=160.0, height=33.0)

canvas\_form.create\_text(500.0, 160.0, anchor="nw", text="Work Type", fill="#000000", font=("Inter", 12 \* -1))

entry\_worktyp = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_worktyp.place(x=600.0, y=160.0, width=160.0, height=33.0)

canvas\_form.create\_text(500.0, 220.0, anchor="nw", text="Branch", fill="#000000", font=("Inter", 12 \* -1))

branches = Process.load\_branches()

branch\_combobo = ttk.Combobox(form\_frame, values=branches)

branch\_combobo.place(x=600.0, y=220.0, width=160.0, height=33.0)

canvas\_form.create\_text(500.0, 280.0, anchor="nw", text="Password", fill="#000000", font=("Inter", 12 \* -1))

entry\_passwor = Entry(form\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0, show="\*")

entry\_passwor.place(x=600.0, y=280.0, width=160.0, height=33.0)

# Error Label

error\_label4 = Label(form\_frame, text="", bg="#E7E8D1", fg="red", font=("Inter", 10))

error\_label4.place(x=250, y=450)

canvas\_form.create\_text(70.0, 350.0, anchor="nw", text="Insert an image\nof you for\nidentification", fill="#000000", font=("Inter", 12 \* -1))

# Add upload and download buttons in profile\_frame

upload\_button = Button(

profile\_frame,

text="Upload File",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:handle\_upload()

)

upload\_button.place(x=250, y=350, width=150, height=40)

download\_button = Button(

profile\_frame,

text="Download File",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda:handle\_download()

)

download\_button.place(x=410, y=350, width=150, height=40)

# Submit Button

submit\_button = Button(

form\_frame,

text="Save Changes",

command=lambda: update\_account\_validation(current\_user, entry\_firstnam, entry\_lastnam, entry\_do, entry\_emai, entry\_addres, entry\_worktyp, branch\_combobo, entry\_passwor),

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

submit\_button.place(x=250.0, y=500.0, width=210.0, height=40.0)

#----------------------------------------------------------------------------------------------

# Send Frame

send\_frame = Frame(window, bg="#D7DEC8")

send\_frame.pack(fill="both", expand=True)

send\_frame.pack\_forget()

# Sidebar (Left Side)

sidebar\_frame\_profile = Frame(send\_frame, bg="#B85042", width=90, height=600)

sidebar\_frame\_profile.place(x=0, y=0)

Button(

sidebar\_frame\_profile,

text="Home",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_home\_page()

).place(x=0, y=19, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Profile",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_profile\_page()

).place(x=0, y=93, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Send",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_send\_page()

).place(x=0, y=167, width=90, height=40)

Button(

sidebar\_frame\_profile,

text="Log out",

bg="#B85042",

fg="#E7E8D1",

font=("Inter", 14),

relief="flat",

command=lambda: switch\_to\_login\_page()

).place(x=0, y=533, width=90, height=40)

# Main Content (Right Side)

canvas\_send = Canvas(

send\_frame,

bg="#D7DEC8",

height=600,

width=900,

bd=0,

highlightthickness=0,

relief="ridge"

)

canvas\_send.place(x=90, y=0)

canvas\_send.create\_text(

300.0,

20.0,

anchor="nw",

text="Send Money",

fill="#B85042",

font=("Inter SemiBold", 24)

)

# Recipient Input

canvas\_send.create\_text(

100.0,

80.0,

anchor="nw",

text="Recipient",

fill="#000000",

font=("Inter", 12)

)

entry\_recipient = Entry(send\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_recipient.place(x=350.0, y=80.0, width=400.0, height=30.0)

# Amount Input

canvas\_send.create\_text(

100.0,

140.0,

anchor="nw",

text="Amount (£)",

fill="#000000",

font=("Inter", 12)

)

entry\_amount = Entry(send\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_amount.place(x=350.0, y=140.0, width=400.0, height=30.0)

# Description Input (Optional)

canvas\_send.create\_text(

100.0,

200.0,

anchor="nw",

text="Description (Optional)",

fill="#000000",

font=("Inter", 12)

)

entry\_description = Entry(send\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_description.place(x=350.0, y=200.0, width=400.0, height=30.0)

# Date Display (Auto-Generated)

canvas\_send.create\_text(

100.0,

260.0,

anchor="nw",

text="Date (YYYY-MM-DD)",

fill="#000000",

font=("Inter", 12)

)

entry\_date = Entry(send\_frame, bd=0, bg="#FFFFFF", fg="#000716", highlightthickness=0)

entry\_date.place(x=350.0, y=260.0, width=200.0, height=30.0)

error\_label5 = Label(send\_frame, text="", bg="#E7E8D1", fg="red", font=("Inter", 10))

error\_label5.place(x=400, y=400)

# Submit Button

submit\_button = Button(

send\_frame,

text="Submit",

bg="#B85042",

fg="#FFFFFF",

font=("Inter", 14),

relief="flat",

command=lambda: send\_validation(current\_user, entry\_recipient, entry\_amount, entry\_description, entry\_date)

)

submit\_button.place(x=400.0, y=320.0, width=150.0, height=40.0)

instructions\_label = Label(

send\_frame,

text="Save the list of all your transactions or import them if you need to restore your data!",

bg="#D7DEC8",

fg="#000000",

font=("Inter", 12),

wraplength=400,

justify="center"

)

instructions\_label.place(x=220, y=400, width=500, height=50)

export\_button = Button(

send\_frame,

text="Export Transactions",

command=export\_transactions\_to\_xml,

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

export\_button.place(x=250, y=480, width=180, height=40)

import\_button = Button(

send\_frame,

text="Import Transactions",

command=import\_transactions\_from\_xml,

bg="#B85042",

fg="white",

font=("Inter", 12),

relief="flat"

)

import\_button.place(x=500, y=480, width=180, height=40)

#-----------------------------------------------------------------------------------------------

# Run the application

window.resizable(False, False)

window.mainloop()

* **Process.py code:**

import sqlite3

import hashlib

from datetime import datetime

import xml.etree.ElementTree as ET

connection = sqlite3.connect(r"Database\NoteBankSystem.db")

cursor = connection.cursor()

# Function to hash user data for security

def hashdata(data):

"""

Hash user data using SHA-256 with added salt for enhanced security.

This ensures data protection even in case of unauthorized access.

"""

data = "!£" + data # Add salt to the data

data\_bytes = data.encode('utf-8') # Convert data to bytes

sha256 = hashlib.sha256()

sha256.update(data\_bytes) # Hash the salted data

return sha256.hexdigest() # Return the hexadecimal representation of the hash

def signin(email\_entry, password\_entry):

"""

Validate user credentials by comparing hashed email and password.

Returns a tuple where the first value indicates success and the second value is the user ID.

"""

cursor.execute("SELECT UserId, Email, Password FROM Users")

rows = cursor.fetchall()

email = email\_entry.get()

password = password\_entry.get()

if not email or not password: # Check if fields are empty

return [False]

email = hashdata(email) # Hash the email

password = hashdata(password) # Hash the password

for row in rows:

if row[1] == email and row[2] == password:

return [True, row[0]] # Return success and user ID

return [False] # Return failure

def create\_account(entry\_firstname, entry\_lastname, entry\_dob, entry\_email, entry\_address, entry\_worktype, EntryBranch, entry\_password):

"""

Create a new user account in the database.

Validates required fields and ensures the branch exists.

"""

firstname = entry\_firstname.get()

lastname = entry\_lastname.get()

dob = entry\_dob.get()

email = entry\_email.get()

address = entry\_address.get()

work = entry\_worktype.get()

branch = EntryBranch.get()

password = entry\_password.get()

tBranch = load\_branches() # Load available branches

if branch not in tBranch: # Check if branch is valid

return False

else:

cursor.execute("SELECT BranchID FROM Branches WHERE BranchName = ?", (branch,))

result = cursor.fetchone()

if not (firstname and lastname and branch and email and password): # Check required fields

return False

# Insert user data into the database

cursor.execute(

'''INSERT INTO Users (FirstName, LastName, DOB, Email, Address, WorkType, BranchID, Password)

VALUES (?, ?, ?, ?, ?, ?, ?, ?)''',

(hashdata(firstname), hashdata(lastname), dob, hashdata(email), hashdata(address), hashdata(work), result[0], hashdata(password))

)

connection.commit()

return True # Account creation successful

def load\_branches():

"""

Retrieve all available branches from the database.

Returns a list of branch names.

"""

cursor.execute("SELECT BranchName FROM Branches")

branches = [row[0] for row in cursor.fetchall()]

return branches

def reset\_password(entry\_firstname, entry\_lastname, entry\_password, entry\_email, entry\_address, entry\_work, branch\_combobox):

"""

Reset a user's password if their credentials match.

Validates all fields and checks for matching account details.

"""

firstname = entry\_firstname.get()

lastname = entry\_lastname.get()

new\_password = entry\_password.get()

email = entry\_email.get()

address = entry\_address.get()

work = entry\_work.get()

branch = branch\_combobox.get()

tBranch = load\_branches()

if branch not in tBranch:

return [True, False]

cursor.execute("SELECT BranchID FROM Branches WHERE BranchName = ?", (branch,))

result = cursor.fetchone()

cursor.execute("SELECT \* FROM Users WHERE Email = ?", (hashdata(email),))

emailCheck = cursor.fetchone()

if not emailCheck: # Email does not exist

return [False, False]

# Check required fields

if not (firstname and lastname and email and address and work and result):

return [False, True]

# Verify account details

cursor.execute(

"""

SELECT \* FROM Users

WHERE Email = ? AND FirstName = ? AND LastName = ? AND Address = ? AND WorkType = ? AND BranchID = (

SELECT BranchID FROM Branches WHERE BranchName = ?

)

""",

(hashdata(email), hashdata(firstname), hashdata(lastname), hashdata(address), hashdata(work), branch)

)

user = cursor.fetchone()

if user: # Account details match

cursor.execute("UPDATE Users SET Password = ? WHERE Email = ?", (hashdata(new\_password), hashdata(email)))

cursor.connection.commit()

return [True, False]

else:

return [True, True] # Verification failed

def balance(user):

"""

Retrieve the current balance and IBAN for a given user.

"""

cursor.execute("SELECT Balance, IBAN FROM Accounts WHERE UserID = ?", (user,))

return cursor.fetchone()

def top\_up(quantity, user):

"""

Increase the balance of a user's account by the given amount.

"""

query = "UPDATE Accounts SET Balance = Balance + ? WHERE UserId = ?"

cursor.execute(query, (quantity, user))

connection.commit()

def insert\_file\_and\_convert\_to\_blob(file\_path, current\_user):

"""

Insert a file as a BLOB into the database for a specific user.

"""

if file\_path:

try:

with open(file\_path, 'rb') as file: # Open file in binary mode

img\_data = file.read()

cursor.execute("UPDATE Accounts SET Image = ? WHERE UserId = ?", (img\_data, current\_user))

connection.commit()

print("File successfully converted to BLOB and inserted into the database.")

except Exception as e:

print(f"Error: {e}")

def download\_file(file\_id):

"""

Download a file (BLOB) from the database for a specific user.

"""

try:

cursor.execute("SELECT Image FROM Accounts WHERE UserId = ?", (file\_id,))

file\_data = cursor.fetchone()

if file\_data:

return file\_data[0] # Return binary data for the file

else:

print("No file found in the database.")

return None

except Exception as e:

print(f"Failed to download the file: {e}")

return None

def upload\_file(file\_path, user):

"""

Handle file upload for a specific user.

"""

if file\_path:

try:

insert\_file\_and\_convert\_to\_blob(file\_path, user)

print("File uploaded successfully!")

except Exception as e:

print(f"Failed to upload the file: {e}")

def update\_account(current\_user, entry\_firstname, entry\_lastname, entry\_dob, entry\_email, entry\_address, entry\_work, branch\_combobox, entry\_password):

"""

Update user account details with the provided information.

Validates all fields and updates only the provided ones.

"""

firstname = entry\_firstname.get()

lastname = entry\_lastname.get()

new\_password = entry\_password.get()

email = entry\_email.get()

dob = entry\_dob.get()

address = entry\_address.get()

work = entry\_work.get()

branch = branch\_combobox.get()

tBranch = load\_branches()

if branch == "":

result = ""

elif branch not in tBranch:

return False # Invalid branch

cursor.execute("SELECT BranchID FROM Branches WHERE BranchName = ?", (branch,))

branch\_result = cursor.fetchone()

result = branch\_result[0] if branch\_result else ""

cursor.execute("SELECT \* FROM Users WHERE UserId = ?", (current\_user,))

userCheck = cursor.fetchone()

if userCheck[0]:

updates = []

params = []

if firstname:

updates.append("FirstName = ?")

params.append(hashdata(firstname))

if lastname:

updates.append("LastName = ?")

params.append(hashdata(lastname))

if new\_password:

updates.append("Password = ?")

params.append(hashdata(new\_password))

if email:

updates.append("Email = ?")

params.append(hashdata(email))

if dob:

updates.append("DOB = ?")

params.append(dob)

if address:

updates.append("Address = ?")

params.append(hashdata(address))

if work:

updates.append("WorkType = ?")

params.append(hashdata(work))

if result != "":

updates.append("BranchID = ?")

params.append(result)

if updates: # Execute the query if there are updates

update\_query = f"UPDATE Users SET {', '.join(updates)} WHERE UserId = ?"

params.append(current\_user)

cursor.execute(update\_query, tuple(params))

cursor.connection.commit()

return True

def transaction(user, entry\_recipient, entry\_amount, entry\_description, entry\_date):

"""

Process a transaction between two accounts.

Validates fields and ensures sufficient balance.

"""

recipient = entry\_recipient.get()

amount = entry\_amount.get()

description = entry\_description.get()

date = entry\_date.get()

if not recipient or not amount or not date:

return 1 # Missing fields

try:

parsed\_date = datetime.strptime(date, "%Y-%m-%d").date() # Validate date format

except ValueError:

return 2 # Invalid date format

amount = float(amount)

Balance = balance(user)

if amount > float(Balance[0]):

print("The amount is too much ", amount)

return 3 # Insufficient balance

Balance = float(Balance[0]) - amount

# Insert the transaction record

query = """

INSERT INTO Transactions (AccountId, Amount, Recipient, Description, Date)

VALUES (?, ?, ?, ?, ?)

"""

cursor.execute(query, (user, amount, recipient, description, date))

connection.commit()

# Update the account balance

cursor.execute("UPDATE Accounts SET Balance = ? WHERE UserId = ?", (Balance, user))

connection.commit()

return 5 # Transaction successful

def list\_transactions():

"""

List all transactions for the current year, grouped by month.

Returns a list of months and their total transaction amounts.

"""

current\_year = datetime.now().year

query = """

SELECT strftime('%Y-%m', Date) AS Month, SUM(Amount) AS TotalAmount

FROM Transactions

WHERE strftime('%Y', Date) = ?

GROUP BY strftime('%Y-%m', Date)

ORDER BY Month

"""

cursor.execute(query, (str(current\_year),))

transactions = cursor.fetchall()

return transactions

def export\_transactions(user\_id):

"""

Export transactions for a user to XML format.

"""

# Fetch transactions from the database

cursor.execute("""

SELECT t.TransactionId, t.AccountId, t.Amount, t.Recipient, t.Description, t.Date

FROM Transactions t

INNER JOIN Accounts a ON t.AccountId = a.AccountID

WHERE a.UserID = ?

""", (user\_id,))

transactions = cursor.fetchall()

# Create the root element

root = ET.Element("Transactions")

for transaction in transactions:

transaction\_element = ET.SubElement(root, "Transaction")

ET.SubElement(transaction\_element, "TransactionId").text = str(transaction[0])

ET.SubElement(transaction\_element, "AccountID").text = str(transaction[1])

ET.SubElement(transaction\_element, "Amount").text = str(transaction[2])

ET.SubElement(transaction\_element, "Recipient").text = transaction[3]

ET.SubElement(transaction\_element, "Description").text = transaction[4]

ET.SubElement(transaction\_element, "Date").text = transaction[5]

# Return the root element for further processing

return root

def import\_transactions\_from\_xml(file\_path):

"""

Import transactions from an XML file into the database.

"""

try:

# Parse the XML file

tree = ET.parse(file\_path)

root = tree.getroot()

# Iterate through transactions and insert them into the database

for transaction in root.findall("Transaction"):

account\_id = transaction.find("AccountID").text

amount = transaction.find("Amount").text

recipient = transaction.find("Recipient").text

description = transaction.find("Description").text

date = transaction.find("Date").text

print("Acc:", account\_id , "AMount:", amount, "REcip: ", recipient, "Descr: ",description, "Date: ", date)

# Insert or ignore duplicate transactions

cursor.execute(

"INSERT OR IGNORE INTO Transactions (AccountID, Amount, Recipient, Description, Date) VALUES (?, ?, ?, ?, ?)",

(account\_id, amount, recipient, description, date),

)

connection.commit()

return True # Success

except Exception as e:

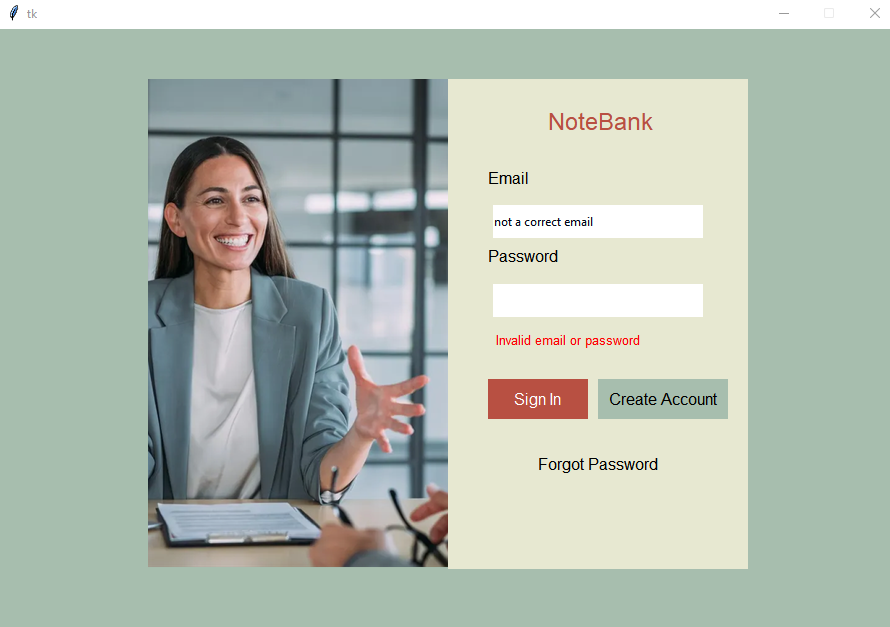
print(f"Error importing transactions: {e}")

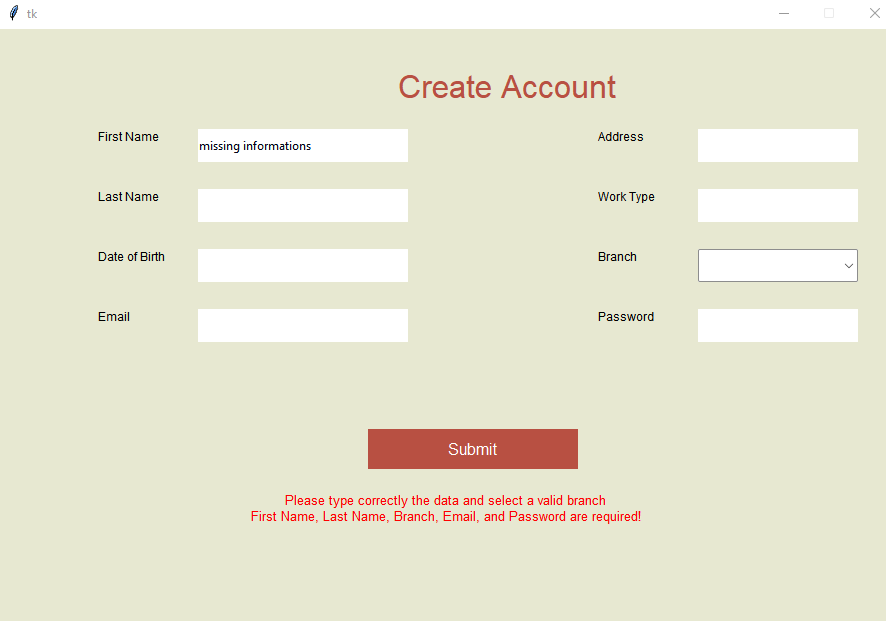
return False # Failure

1. **Verification of data input**

To ensure that the data prompted from the user during the execution of NoteBank the program automatically checks the correctness and if something isn't correct error messages will be displayed to inform the user.

Examples in all the pages of the program:

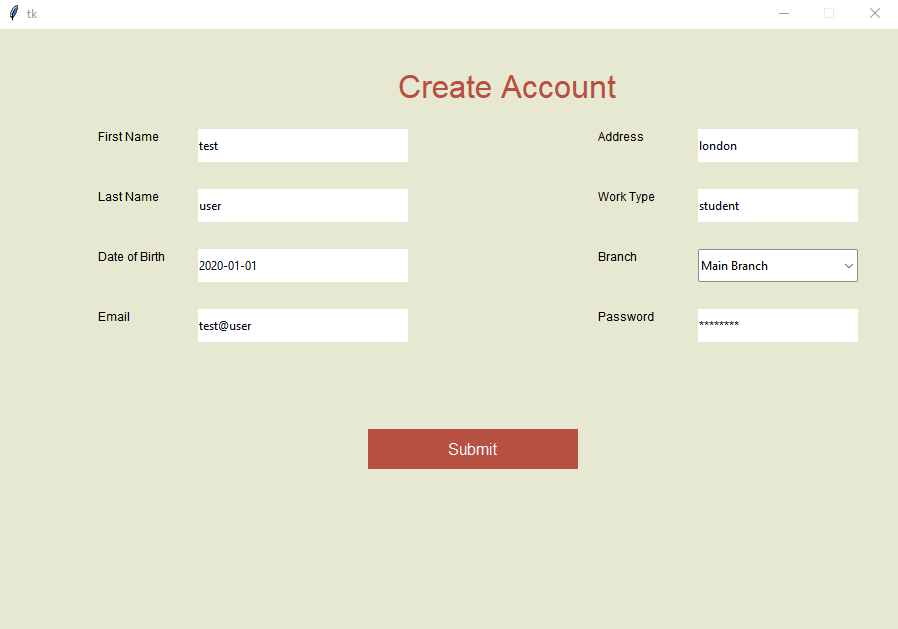




1. **Encryption**

In a bank system security is the top priority, particularly when handling sensitive user data such as passwords. To ensure data security, the program applies encryption techniques to protect the data of the users and sensitive information. Specifically, the **SHA-256 hashing algorithm** is used to securely hash the data. SHA-256 is a cryptographic hash function that converts input data into a fixed length hash, making it nearly impossible to reverse the original data. This approach ensures that even if the database is compromised, user passwords remain secure, as the hash cannot be easily decrypted. Also since the hashes are almost impossible to be reversed to check the user data as email or password we just check if the hash of the imput corrisponds to the one inside the database.

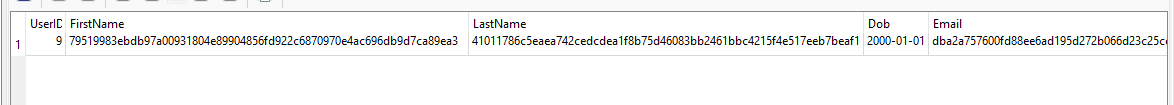
We have previously created an account in the program inside the ‘create account’ page:



Retriving the data from the database with the query:

SELECT \* FROM Users WHERE UserID = 9;

Will show us that the inserted data have been hashed:



The hashing process happens in the Process.py file with the function:

Import hashlib

def hashdata(data):

"""

Hash user data using SHA-256 with added salt for enhanced security.

This ensures data protection even in case of unauthorized access.

"""

data = "!£" + data # Add salt to the data

data\_bytes = data.encode('utf-8') # Convert data to bytes

sha256 = hashlib.sha256()

sha256.update(data\_bytes) # Hash the salted data

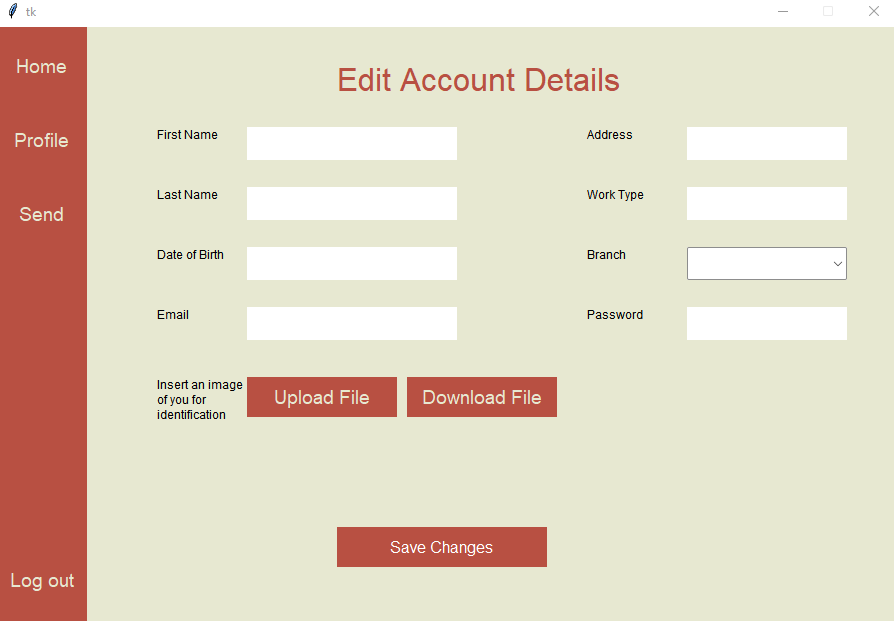
return sha256.hexdigest() # Return the hexadecimal representation of the hash

1. **Storing images in Sql databse with BLOB data type**

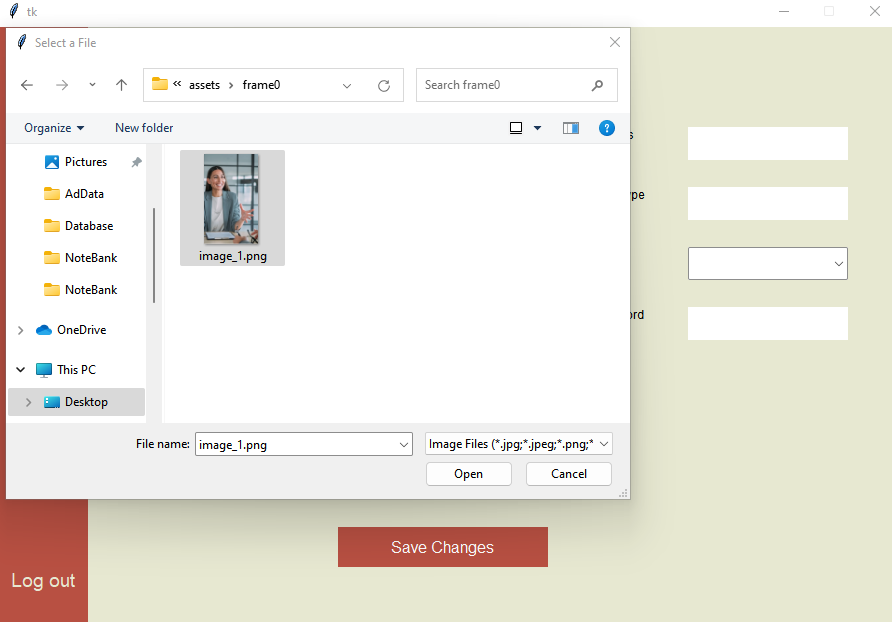
In the NoteBank program, the application uses **BLOB (Binary Large Object)** data type to store and retrieve images in the database. This feature allows users to upload an image linked to their account, such as a profile picture or an ID document. The image is stored directly in the SQLite database in binary format, ensuring that it is securely managed.

The image functionality is inside the ‘profile’ page where the users can manage and change their personal information.

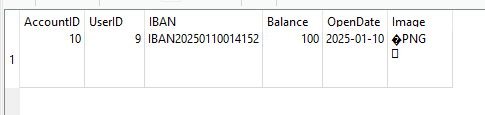
Using always the test@user account that we created before we can see the page provides two buttons, one to insert and the other one to retrieve an image:



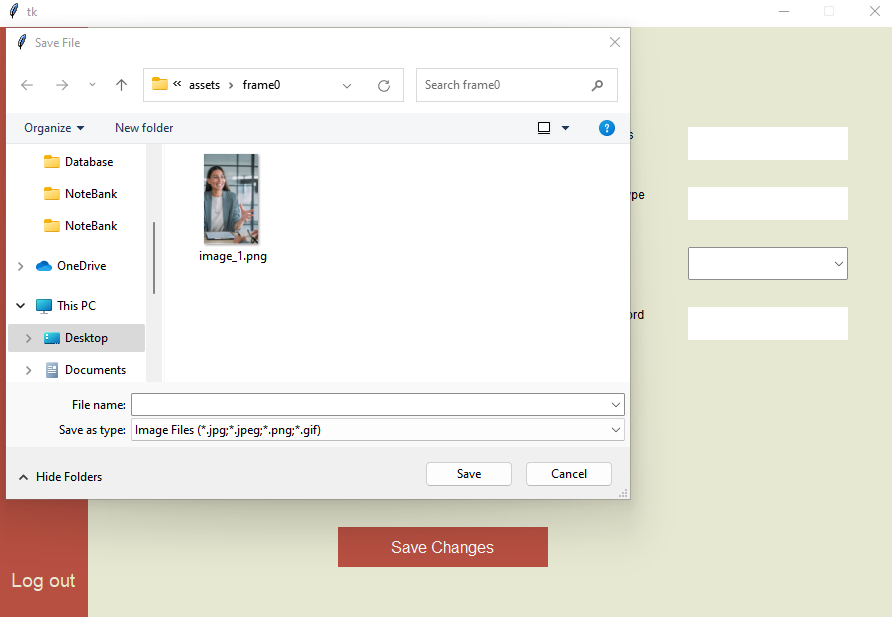
Clicking the upload button the program will ask which file we want to insert:



After we insert an image we can see the database now contains that image stores as a BLOB data type:



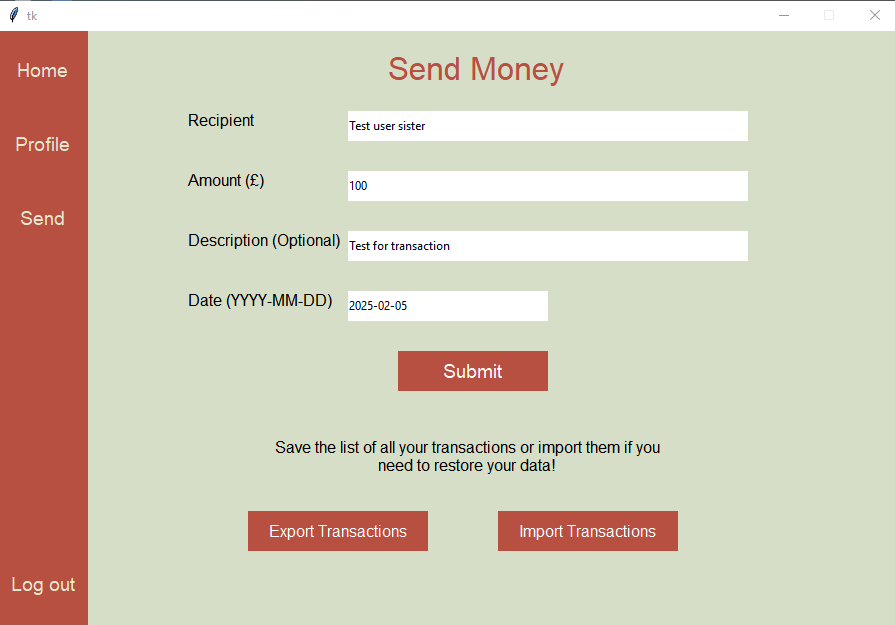
After the upload file we can test also the Download file button and select where the user wants the photo to be saved applying to it a new file-name:



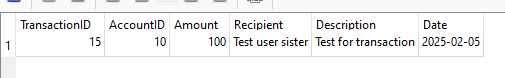
1. **Developing XML**

The XML functionality in the program allows exporting user transactions to a structured XML file. This ensures users can keep a secure backup of their financial records. The program uses python's xml.etree.ElementTree module to generate XML and write it to a file. The program also allows to restore the backup into the database in case the transactions are lost.

Previously a transaction have been inserted with the test@user account:

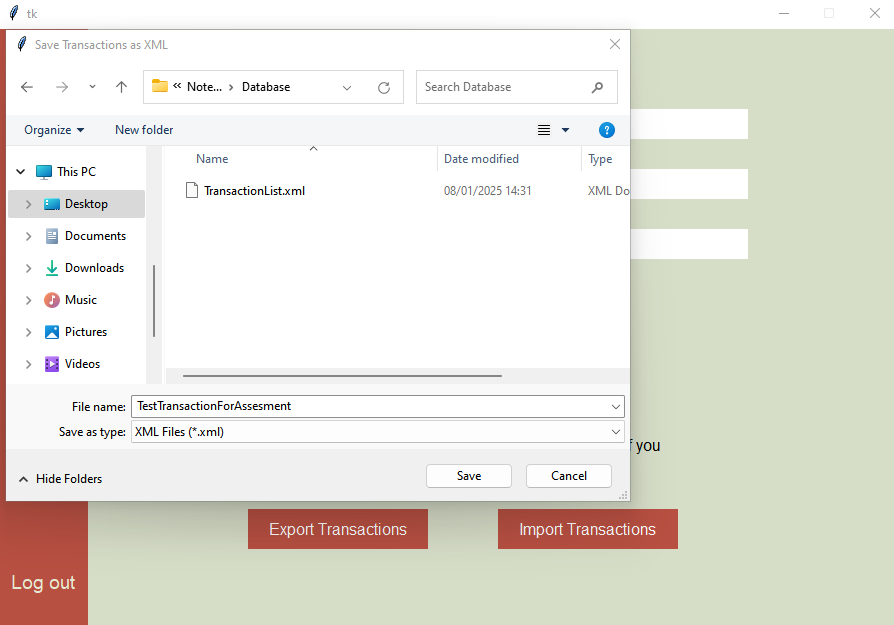


The transaction can be retrived from the database:

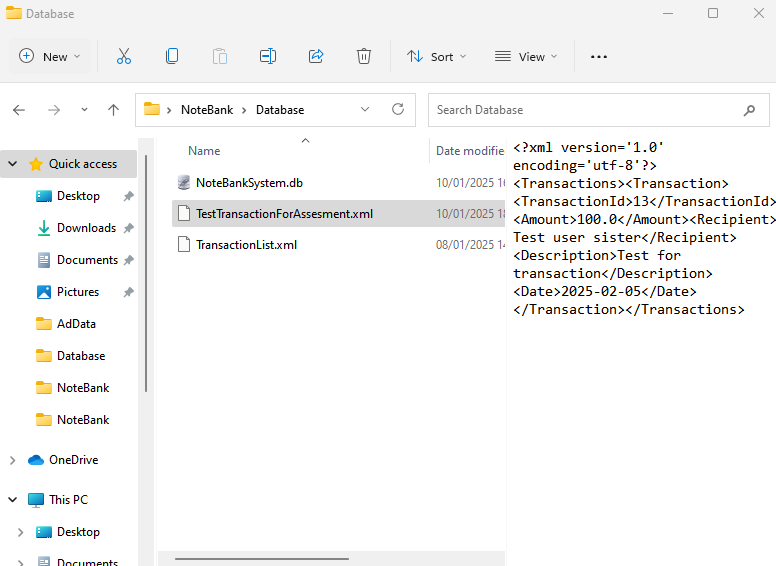


Now on the Send page we can interact with the buttons Export transactions and Import transactions that performs the equivalent action.

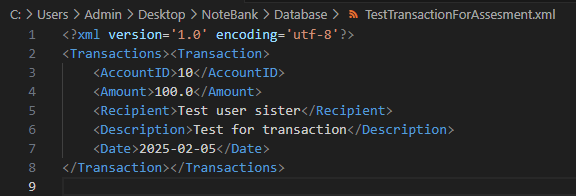
Clicking on export transaction the program will ask to the user what name to give the xml file and in what location to download:



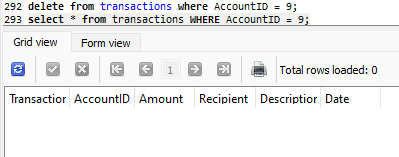
Now in the selected folder we can see the presence of the xml file:



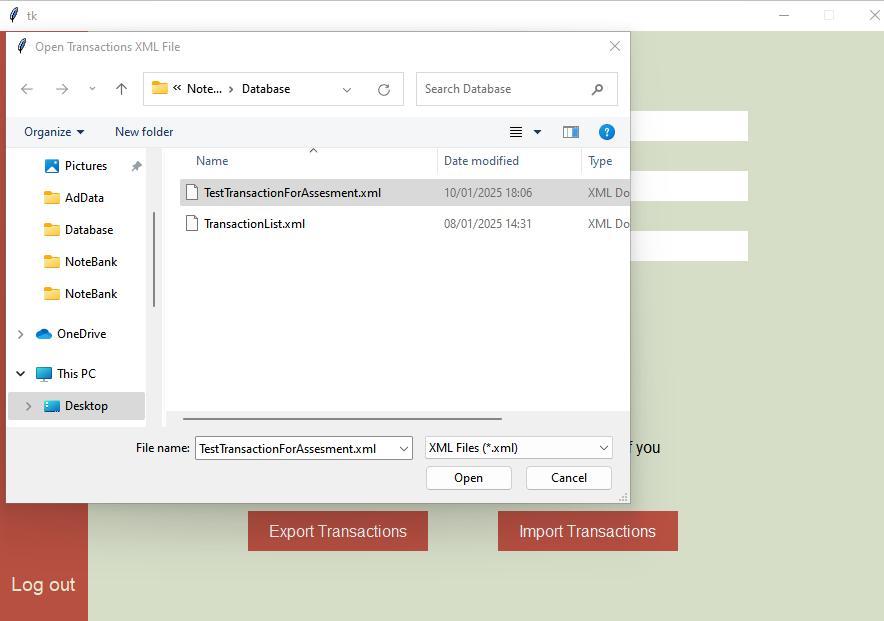
Opening the TestTransactionForAssesment.xml file will show the transaction done by test@user previously:



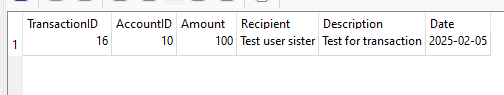
To ensure that the xml implementation works correctly the transaction will be removed from the database and than in the program will run the import transaction functionality.



When clicking the import transaction the user will select the file to import and than the database will be restored :



Now displaying again the information from the database the transaction will be again inserted:



1. **Innovation**

To ensure a level of security, the NoteBank program incorporates two key innovations: the **salt method** for password hashing and measures to prevent **SQL injection attacks**.

#### **Salt Method for Password Security**

The system uses the **SHA-256 hashing algorithm** combined with a unique salt for each password. The salt is a value added to the password before hashing. This method enhances security by protecting against rainbow table attacks, as pre-hashed tables become ineffective. The salt and hash are stored in the database, and the system validates passwords by combining the salt with the input and hashing it for comparison.

Demonstration:

def hashdata(data):

"""

Hash user data using SHA-256 with added salt for enhanced security.

This ensures data protection even in case of unauthorized access.

"""

data = "!£" + data # Add salt to the data

data\_bytes = data.encode('utf-8') # Convert data to bytes

sha256 = hashlib.sha256()

sha256.update(data\_bytes) # Hash the salted data

return sha256.hexdigest() # Return the hexadecimal representation of the hash

Now if we try to hash the word ‘hello’ as an example the result with and without the salt will be completely different:

SHA”%&(“!£! + ‘hello’) = 6347d5a6498a3f5368fcaa33a83112dd62b1ba24c949c97b16d58f59f327a15a

SHA(‘hello’) =

2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

#### **SQL Injection Protection**

To safeguard the database against unauthorized access, the system implements **parameterized queries** that separate SQL logic from user-provided data. This ensures user inputs are treated strictly as data, not executable code, preventing malicious SQL commands. These measures ensure the integrity of the database and protect sensitive information, such as user credentials.

1. **Conclusion**

The NoteBank program and database demonstrates a comprehensive and secure approach to managing user accounts and financial operations. Through features such as secure login, data encryption using the salt method, protection against SQL injections, and functionalities like XML export and BLOB image storage, the system ensures data security and usability.

Every feature was thoroughly tested to confirm accuracy, meeting the requirements for reliability, scalability, and user-friendliness. The integration of advanced security practices and user centered functionalities highlights the system's readiness for real-world application, ensuring that it is both practical and secure.