IM-CW1

Bank Database Report

Student ID: 2136685

Table of Contents

Table of Figures

Entity-Relationship Diagram

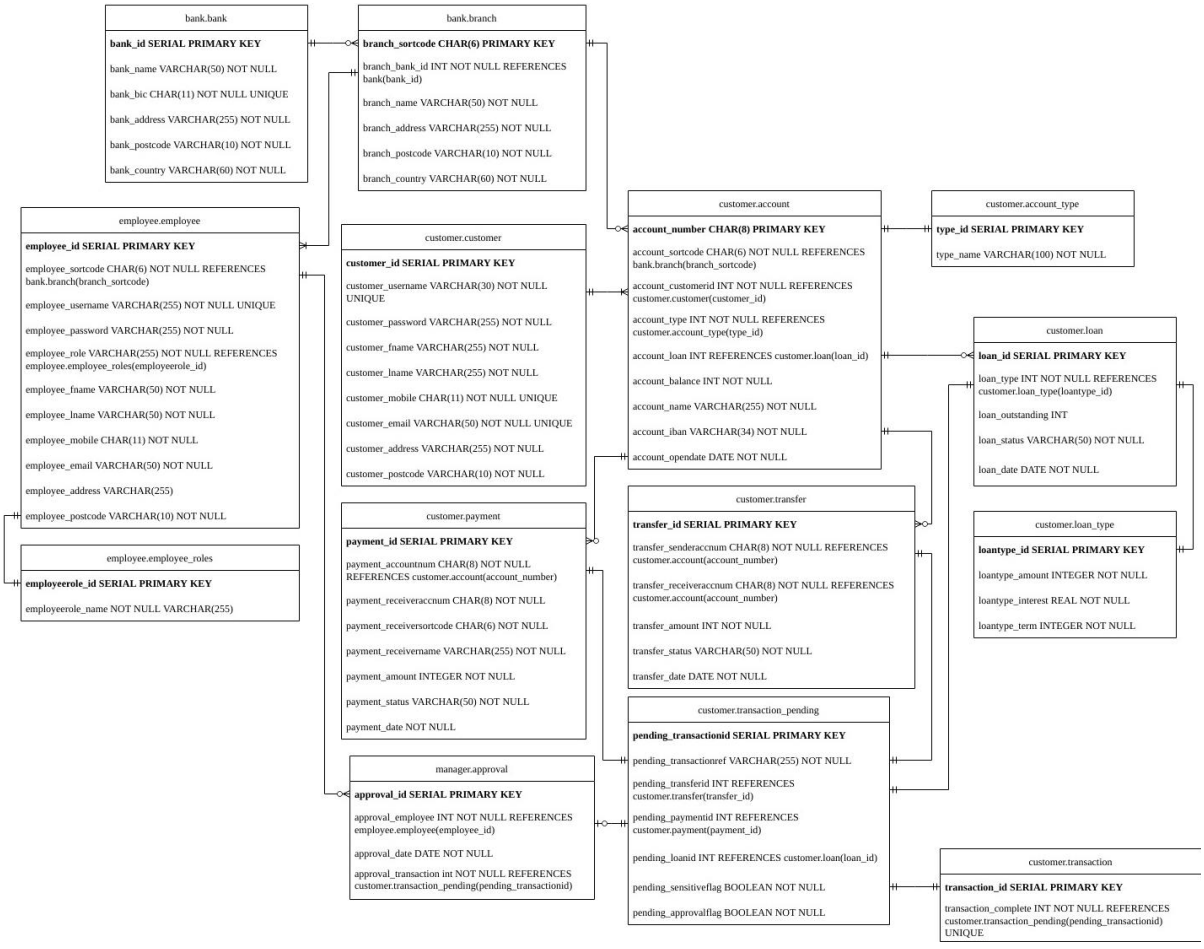

Figure 1 - Entity-Relationship Diagram

My entity-relationship diagram consists of 13 entities. The bold attributes, mostly IDs, are the primary keys of the entities. This is a pattern followed by most of the entities, except for the branch and account entities. These have the “sort_code” and the “account_number” attributes as their primary keys respectively, as they are unique for each of the entities. Other entities such as “employee.employee” and “customer.customer” have other unique attributes such as their “username” one, however they are not used as primary keys.

Some attributes are allowed up to 255 characters, such as the address attributes, however some have more specific restraints. For example, the “account_iban” attribute in the “customer.account” entity has a varchar limit of 34 characters, as IBANs are a maximum of 34 characters, but they can be less. Another example would be the “customer_postcode”. The maximum limit of characters for this attribute is 11, however some postcodes might be less than 11, but they are never more than 11 in the UK. On the other hand, the “customer_mobile” field must be exactly 11 numbers, as that is standard for a UK number.

The most logical way was to start from the root of the bank system, therefore I started with the bank table. In my test script, I used Lloyds Bank as a bank in my system. I implemented this to have support for banks that have international branches as well, for example Barclays. The bank entry also has a unique “bank_bic” field, which is the unique bank identifier code, complying with the ISO 9362:2022 standard. A bank can have multiple branches, so each branch gets a unique sort code attribute and an attribute referencing the bank ID of the bank it belongs to. (ISO, 2022)

The customer table holds all personal details of a customer that wishes to create an account with the bank, such as their full name and address. A customer can only associate their mobile number with one account, as the “customer_mobile” attribute is unique to each customer – this applies to the “customer_email” attribute as well. A customer applying for an account would get a unique customer ID, making the entry easily accessible and referenceable. A customer must have a way to log into the system and access their account, therefore every customer must provide a username and password, where the username will be unique to them, so their account can be linked to their username. If the front-end complies with GDPR, the password provided for the entity entry should be hashed with a standard and modern procedure, a procedure which I will discuss more in-depth in the user journey section.

Every customer can apply for a bank account, be it their first bank account or an extra bank account. Bank accounts are uniquely recognized through their account number which would be generated by the front-end of the system, however in my implementation any unique eight-character number can be inserted, as implementing an account number generator is out of the scope of the assignment. Each bank account is associated with a customer account using their customer ID, which is set when creating an account. All accounts must have an account type, be it current account or a savings account, therefore the “customer.account_type” entity is created to keep the database normalized.

A bank account can have a myriad of transactions, such as payments, transfers, or loans. Therefore, I created entities for each of those types of transactions, along with a “customer.transaction_pending” entry to hold all transactions even after they have been approved so they can be referenced to in the future. Transactions such as transfers or payments may be approved by default and entered in the “customer.transaction” table, which references the “customer.transaction_pending” table. They are entered in the former only if the pending transactions are not sensitive, or if they are sensitive but approved. Loans are a type of transaction that needs further approval from a manager so they will be pending until a manager can approve it.

Customers can apply for a loan per bank account. Customers can usually set the APR, amount of money and the term, however for testing purposes, I reduced the amount of loans to 2. Both with different amounts, interest rates and terms. You cannot hold a percentage in PostgreSQL directly, however you can turn it into a decimal number. Therefore, I gave the “loantype_interest” attribute in the “customer.loan_types” entity the “real” attribute type. This is important later for the calculation of the “loan_outstanding” attribute in the “customer.loan” entity, which is the amount of money a customer must pay back.

Customers can make payments to external bank accounts, therefore a payment entry in the entity would take a receiver’s account number, sort code and name, and the data will be sent to or retrieved by a payment system. Implementing such a system would be out of the scope of the assignment, therefore no other entities related to a customer’s external payment were made.

Customers must be able to make loan payments. The loan payments use the same payment entity as explained above; however, the receiver’s details will be zero and the name of the receiver will be “LOAN PAYMENT”.

Each branch must have employees. Each employee is uniquely identified with an employee ID and attached to their specific branch with its sort code. Employees get set their specific roles when they are recorded into the system, so the “employee.employee” entity may hold managers, tellers, and other roles. However, for the scope of this assignment, the only special employee that has more privileges than a normal employee would be the manager, who can approve sensitive transactions. The manager can be identified by checking the “employee_role” field which references the “employeerole_id” attribute in the “employee.employee_roles” entity. All employees must have a username and password that should comply with the bank’s policies, for example they might need to be at least ten characters in length, however this would be checked by the front-end of the system, as this is out of the scope of the assignment. The storing of the password can be compliant with GDPR if the front-end ensures that it is hashed before being passed to the function that creates an employee, as I have mentioned previously when talking about the customer password. When approving a pending transaction, a record of the approval gets stored in the “manager.approval” entity consisting of the manager that approved the transaction, the date and the transaction that was approved.

User journey

In the table below I will explain all my functions, their parameters, and their return values, in the order they are in the "banking.sql" database dump. The parameter labels are self-explanatory, and they are used for the attributes that their labels suggest. If that is not the case, I will explain that attribute further in the description of the function. Moreover, all the return queries in the functions provide feedback by returning a table showing all the details entered, except passwords, to confirm that the action has taken place and that the details entered were the same as the parameters passed. If a function's return query is different, I will explain it further in the description of the function.

Function	Description	Reference
Creating a customer	A customer would first create an account with the bank. The front-end will collect their personal details and pass them through the "bank.create_customer()" function, only available to the bank schema, the only schema that has functions that can modify the customer related entities. The function takes the customer's username, password, first name, last name, mobile number, email, address, and postcode as parameters and inserts it into the customer table in that specific order. The next unique customer ID is assigned to the customer by calling the function "nextval('customer.customer_customer_id_seq')", where the parameter is the customer ID sequence. This imitates the registration form of a bank registration page, with some limitations. As an example, my implementation stores the password in plain text for testing purposes. Per standard procedure, passwords would be hashed with a random cryptographic salt, however that is out of this assignment's scope, and the password hashing is usually handled by the front-end. For the bank's system to comply with GDPR, the password must not be stored in plain text and as I have described previously. The topic of GDPR will be discussed more in-depth in the GDPR chapter below. (ICO, 2019a)	**Figure 2**
Creating a bank account	A customer needs to have a bank account, regardless of whether they are a new customer or an existing customer. The function "bank.create_account()" creates an account for an existing customer, only available to the bank schema. The function takes the username of the customer that will be associated with the account, account number, sort code, account type and the account name as parameters. Usually, the front-end of the system would have an algorithm to generate an account number and passed to the function, however in my implementation, any account number can be passed if it is specifically eight characters, the standard in the UK. The IBAN variable is generated from the prefix 'GB73LOYD', the sort code and the account number. As I am using a variable, the Dynamic SQL code must be run with the "EXECUTE" command to be able to use the variable. (PostgreSQL, 2022) For testing purposes, the function gives any new account the initial balance of £3000. This is because implementing a function that allows customers to deposit money into their accounts is out of the scope of the assignment.	**Figure 3**

Checking customer details	A customer should be able to check their details. The "customer.check_customer()" takes a customer ID as a parameter to identify a specific entry in the "customer.customer" table. The function returns all the details of the customer entry, without the password or username. If the customer ID does not exist in the customer table, the function returns "Customer does not exist." to alert the front-end that the customer ID is wrong.	**Figure 4**
Checking bank account details	The "customer.check_accounts()" function returns all the bank accounts associated with a customer ID. The function returns everything about each bank account, however instead of showing the ID for the type of bank account, it displays the name. This was done by joining the "customer.account_type" table on the "account_type" foreign key in the "customer.account" table. If the customer ID does not exist, the function raises an error, alerting the front-end that the customer does not exist.	**Figure 5**
Checking accounts' balances	A customer can check all their accounts' balances if the "customer.check_balances()" function is run. The function takes one parameter, this being the customer's ID that would be currently logged in. This would be passed by the front-end using a session handler to get the customer ID, however implementing this is out of the scope of the assignment, so I assume that the customer ID would be correctly passed to the function. The function returns the balances along with the account numbers, names and the account type names about all a customer's accounts. If an error occurs and the customer ID is not found, the function will return a "Customer does not exist." error instead.	**Figure 6**
Checking transfers	The "customer.check_transfers()" function takes a customer ID as a parameter. It returns all the details from the "customer.transfer" table and the transaction ID associated with the transfer and the transaction reference from the "customer.transaction_pending" table. The "#variable_conflict use_column" line ensures that if there are variables and column names sharing the same names, it assumes that the second mention is referencing a column. For example, the return variable "transfer_id" and the "tr.transfer_id" in the JOIN statement. The return query statement works as it looks for the account numbers associated with the customer ID, as seen in the "WHERE" line of the return query.	**Figure 7**
Checking payments	The "customer.check_payments()" works in the same way as the "customer.check_transfers()" function I explained above. Instead of checking the "customer.transfer" table, it checks the "customer.payment" table and returns all its columns.	**Figure 8**
Checking loans	A customer must be able to check their loans. The function "customer.check_loans()" takes a customer ID as a parameter. The function returns a table consisting of the transaction ID, loan ID, the account number associated with the loan, the amount, the outstanding balance, the interest rate, the term, the status, and the date. The outstanding balance will only be set if the status of the loan is "ACTIVE" and not "PENDING". The outstanding balance is calculated with a simple interest for testing purposes, as it is out of the scope of the assignment to calculate an accurate outstanding balance with industry fees and compound interest.	**Figure 9**

Creating a new employee	Creating a new employee is alike creating a customer, however it also takes the branch's sort code and the role of the employee as a parameter. The "bank.create_employee()" function inserts a new employee into the "employee.employee" table. It associates the employee with the branch they work at with the "employee_sortcode" attribute, which is a foreign key referencing the "bank.branch" "branch_sortcode" primary key. Their role is set by the foreign key constraint "employee_role" in the "employee.employee" table from the "employee.employee_roles" table. The roles were separated to ensure database normalisation. The function returns all the employee's details, without the password and username. As explained in the "creating customer" section above, the password is stored in plain text for testing purposes, as it is out of the scope of the assignment to store it otherwise.	**Figure 10**
Check employee details	Employees and managers can check employee details. The function "employee.check_employee()" takes the employee ID as a parameter and returns all the details about an employee, except the username and password. The function also returns the branch and the role of the employee, which was done by joining the "bank.branch" and "employee.employee_roles" tables. However, if the employee ID does not exist, the function raises an exception where it alerts the front-end that the "Employee does not exist.".	**Figure 11**
Apply for loan	Applying for a loan only takes two parameters, the account number a customer will want the loan in and the type of loan they want to get. The "bank.apply_loan()" function returns the loan ID, loan type, status, date, loan type ID, amount, interest, and term. The function ensures that the account number exists. If so, it will insert a "PENDING" loan with the details of the loan chosen into the "customer.loan" table. The function updates the "customer.account" table, specifically the row that holds the account number from the parameters and sets the account_loan attribute to the current loan ID. The loan transaction is inserted into the "customer.transaction_pending" table, setting the status to "PENDING", the sensitive flag to true and the approval flag to false. The flags will show the manager that the transaction needs approval. If the account number does not exist in the "customer.account" table, the function will return an error saying that the "Account does not exist". The return query selects all the loan data, without the outstanding balance, where the loan ID is the current ID in the "loan_id" sequence.	**Figure 12**
Making a payment	Customers can make payments to external accounts however the external handling of the payment is not implemented as it is out of the scope of this assignment. The function takes in the data required for an external payment system and it is assumed that the external payment system will handle it. The "bank.make_payment()" function takes the sender account number, the receiver account number, sort code and name, and the amount of the payment. A payment cannot be below 0, so a customer cannot send a negative amount of money to someone. The function ensures that the sender account is correct, otherwise it outputs the "sender account does not exist error". If the amount for the payment is more than the balance in	**Figure 13**

	the bank account, the function errors. If all the conditions are met, the money is taken out of the bank account and the payment details from the function's parameters are inserted into the "customer.payment" table, along with the payment status being set as "COMPLETE" and the current date. The transaction is also inserted into the "customer.transaction_pending" table, setting the sensitive flag to false and the approval flag to true, because payment transactions do not need manager approval. The pending transaction is inserted into the "customer.transaction" table to show that the transaction has been completed, using the current payment_id sequence to give the foreign key ID. The function returns all the details about the current payment into a table.	
Making a loan payment	The function "bank.pay_loan()" takes an account number and an amount to pay towards the outstanding balance of the loan as parameters. Making a loan payment is alike making a normal payment. It inserts data into the same tables as the normal payment function and has the same checks, however the loan payment function also checks if the amount paid back is more than the outstanding balance of the loan, if so it errors saying "you cannot pay more than the outstanding balance." The function also checks if the account associated with the account number has a loan set in the "account_loan" attribute, otherwise it errors saying "no loan to pay back". If the "loan_status" attribute in the "customer.loan" table of the loan associated with the account number is "PENDING" then the function errors with "loan is pending. You cannot pay back now.". If the payment passes these checks, the data inserted into the "customer.payment" table is the same as a normal payment, but the receiver's data is replaced by 0s and the receiver's name is "LOAN PAYMENT" instead.	**Figure 14**
Making a transfer	The function "bank.make_transfer()" takes the sender account number, the receiver account number, and the amount to be transferred. The validation checks are the same as the two above functions, however there is one more big check that ensures that the transfer is carried on with only if the sender and receiver account numbers belong to the same customer, otherwise it errors. The function updates the sender account by taking away the amount from the sender's balance and updates the receiver account by adding the amount to the receiver's balance. The transfer does not require further approval so the transfer table, the transaction_pending and the transaction table are all inserted with data like the two functions above. The function returns all the details about the current transfer from the "customer.transfer" table.	**Figure 15**
Manager approving a transaction	The "manager.approve_pending()" function takes the employee ID and the transaction ID as parameters. The function only begins if the employee ID has the role of 'Manager'. If so, checks are done to see if the transaction inserted exists. If it does not, the function errors and outputs "No transaction found". If it is found but the sensitive flag is set to false and the approval flag is set to true, then the function errors and outputs "Transaction is not sensitive or does not need approval". If the transaction passes the checks, then the "customer.transaction_pending" table is updated and the transaction's flag is set to true. If the transaction is a	**Figure 16, Figure 17**

	transfer, the relevant transfer ID is used to update the “customer.transfer” table and set its status to “COMPLETE”. If the transaction is a payment, it does the same as the transfer check but with the payment ID. If the transaction is a loan, the “loan_status” attribute is set to “ACTIVE” for the relevant loan ID. The “customer.loan” table is updated and the “loan_outstanding” attribute is updated with the loan amount plus the interest. The account balance of the account number associated with the loan ID is updated with the amount of the type of loan selected. The “manager.approval” record is inserted into with the next approval ID, the current date of the approval and the transaction ID from the parameters. The “customer.transaction” table is also inserted into to show that the transaction is completed.	

Security practices and GDPR

In order to make my database more secure, I introduced methods such as implementing schemas, roles and users, keeping the principle of least privilege in mind. To secure the whole PostgreSQL process, I changed the “pg_hba.conf” and “postgresql.conf” files to tighten the user access to the databases, enabled SSL in order to use an encrypted TCP/IP connection, disabled IPv6 connections, specified a small amount of users that are allowed to access the “banking” database and only from a specific address. To comply with international standards, I enabled process logs and database connection logs to make a compilation of logs, which are required in security audits by standardization bodies to certify compliance.

To secure the database, roles were only given just enough privilege so they can carry out their specific activity. For testing purposes, there are only 4 schemas in the database, whereas a real bank system would be much more granular with their permissions and schemas. The schemas are as follows:

- “customer” schema – The customer schema can use functions that can only retrieve data from the tables in the customer schema. The schema can be accessed by users in the “role_customer” role, which has been assigned to the user “user_customer1”, as shown in figure 19. The users assigned to the “role_customer” role will get an error when trying to access a function they are not authorised to use, as seen in figure 37.
- “employee” schema – The employee schema holds functions that a customer should not be able to see, such as “employee.check_employee()” which can see the details of an employee. This can be used for a register, for example. The role “role_employee” is given access to this schema, along with the “role_customer” role, to be able to check balances, bank accounts, customer details and loans.
- “manger” schema – The manager schema has a function that can approve sensitive transactions, such as loan applications. The “role_manager”, “role_employee” and “role_customer” roles are assigned to the user “test_manager1”.
- “bank” schema – The bank schema holds all of the functions that alter tables. This is to ensure that not any employee or customer can, for example, change their own balance.

The database only collects the necessary information for the opening of a bank account and nothing more. Moreover, the data collected is extremely sensitive as it includes records such as full names and addresses, so it must be secured with the highest level of security, which is why I have changed the "pg_hba.conf" file to restrict access to PostgreSQL users and to comply with GDPR, as seen in figure 35. The host connection used is "hostssl" which tells PostgreSQL to use an encrypted connection for the database using SSL. Moreover, I specified users that can access the "banking" database to ensure that only they can access it, as they are needed for the test script to function, but only if they are logging in from the localhost address. By default, all users are allowed to connect from all IPv6 addresses without passwords. This is a great security risk, so I disabled it to reduce the attack surface. The "pg_hba.conf" comes default with no password needed for a connection, but it suggests that users' passwords should be encrypted with either "md5" or "scram-sha-256". MD5 is an outdated method, therefore I chose sha-256 for all passwords to ensure maximum security. (ICO, 2019c; Kost, 2023; PostgreSQL Group, 2022b)

For real banks, customer data would be deleted after they have closed their bank account, however that is out of the scope of this assignment, therefore it is not implemented. (ICO, 2019d)

Keeping the data storage and the data transfer encrypted is required for ensuring security and compliance with GDPR. Otherwise, the ICO recognizes that without encryption, there can be serious consequences and the CIA triad will be broken. Without encryption, the confidentiality and integrity elements of the CIA triad would be instantly broken, as both of those would simply not exist. Implementing encryption for storing the data is out of scope for the assignment, however for the data transfer, as mentioned above, I encrypted the TCP/IP connection using SSL. The connection was encrypted using a self-signed certificate and private key as a proof of concept, as seen in figure 36. In a real bank, a certificate authority would be used to show the authenticity of the certificate. (ICO, 2019a; ICO, 2019d; PostgreSQL Group, 2022a)

Standards such as the ISO 27001 require log records in order to get certified, therefore the logs must be enabled, as seen in figure 36. However, they are also helpful for recognizing database access and for debugging. (International Organization for Standardization, 2013)

References

ICO (2019a). *Encryption*. [online] Ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/security/encryption/ [Accessed 29 Jan. 2023].

ICO (2019b). *Passwords in online services*. [online] Ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/security/passwords-in-online-services/ [Accessed 25 Jan. 2023].

ICO (2019c). *Principle (c): Data minimisation*. [online] Ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/principles/data-minimisation/ [Accessed 29 Jan. 2023].

ICO (2019d). *Principle (e): Storage limitation*. [online] Ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/principles/storage-limitation/ [Accessed 29 Jan. 2023].

ICO (2019e). *Security*. [online] Ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/security/ [Accessed 29 Jan. 2023].

International Organization for Standardization (2013). *ISO/IEC 27001 Information security management*. [online] ISO. Available at: https://www.iso.org/isoiec-27001-information-security.html [Accessed 29 Jan. 2023].

ISO (2022). *ISO 9362:2022*. [online] ISO. Available at: https://www.iso.org/standard/84108.html [Accessed 25 Jan. 2023].

Kost, E. (2023). *10 Step Checklist: GDPR Compliance Guide for 2022 | UpGuard*. [online] www.upguard.com. Available at: https://www.upguard.com/blog/how-to-be-gdpr-compliant [Accessed 29 Jan. 2023].

PostgreSQL (2022). *36.5. Dynamic SQL*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/ecpg-dynamic.html [Accessed 25 Jan. 2023].

PostgreSQL Group (2022a). *19.9. Secure TCP/IP Connections with SSL*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/ssl-tcp.html [Accessed 29 Jan. 2023].

PostgreSQL Group (2022b). *21.1. The pg_hba.conf File*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/15/auth-pg-hba-conf.html [Accessed 29 Jan. 2023].

Appendix

```
-- customer can open an account
CREATE OR REPLACE FUNCTION bank.create_customer(param_uname varchar(30), param_pass varchar(30),
param_fname varchar(50), param_lname varchar(50), param_mobile char(11), param_email varchar(50),
param_address varchar(255), param_postcode varchar(10))

RETURNS TABLE (username varchar(30), first_name varchar(50), last_name varchar(50), mobile char(11),
email varchar(50), address varchar(255), postcode varchar(10)) AS $$

BEGIN
    INSERT INTO customer.customer (customer_id, customer_username, customer_password, customer_fname,
    customer_lname, customer_mobile, customer_email, customer_address, customer_postcode)

    VALUES(nextval('customer.customer_customer_id_seq'),param_uname, param_pass, param_fname,
    param_lname, param_mobile, param_email, param_address, param_postcode);

  RETURN QUERY
  SELECT customer_username, customer_fname, customer_lname, customer_mobile, customer_email,
  customer_address, customer_postcode

  FROM customer.customer
  WHERE customer_id = currval('customer.customer_customer_id_seq');

END;
$$ LANGUAGE plpgsql;
```

Figure 2 - Function for opening an account for customers

```
-- existing customer can open another account and new customers can open account first account
CREATE OR REPLACE FUNCTION bank.create_account(param_username varchar(30),
param_accountnum char(8), param_sortcode char(6), param_accountype int, param_accountname varchar(255))

RETURNS TABLE(first_name varchar(50), last_name varchar(50), acc_number char(8),
sort_code char(6), type int, balance int, name varchar(255), iban varchar(34), date date) AS $$
BEGIN

    DECLARE
        param_iban VARCHAR(50);
    BEGIN
        param_iban := 'GB73LOYD' || param_sortcode || param_accountnum;
    EXECUTE 'INSERT INTO customer.account(account_number, account_sortcode, account_customerid,
    account_type, account_balance, account_name, account_iban, account_opendate)
    VALUES ($1, $2, $3, $4, 3000, $5, '''|| format(param_iban) ||''', NOW())'

    USING param_accountnum, param_sortcode, (select customer_id from customer.customer
    where customer_username = param_username), param_accountype, param_accountname, param_iban;
    END;

    RETURN QUERY
    SELECT c.customer_fname, c.customer_lname, a.account_number, a.account_sortcode, a.account_type,
    a.account_balance, a.account_name, a.account_iban,a. account_opendate

    FROM customer.account a
    JOIN customer.customer c ON a.account_customerid = c.customer_id
    WHERE account_number = param_accountnum;

END;
$$ LANGUAGE plpgsql;
```

Figure 3 - Function for opening a bank account for all customers

```
-- customer can check their details
CREATE OR REPLACE FUNCTION customer.check_customer(id int)
RETURNS TABLE (first_name varchar(50), last_name varchar(50), mobile char(11),
email varchar(50), address varchar(255), postcode varchar(10)) AS $$
BEGIN
  -- return customer details for the table
  RETURN QUERY
  SELECT customer_fname, customer_lname, customer_mobile, customer_email,
  customer_address, customer_postcode

  FROM customer.customer
  WHERE customer_id = id;

-- if the customer does not exist
    IF NOT EXISTS (SELECT 1 FROM customer.customer WHERE customer_id=id) THEN
        RAISE EXCEPTION 'Customer does not exist';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Figure 4 - Function for checking customer details

```sql
-- customer can see their bank account details
CREATE OR REPLACE FUNCTION customer.check_accounts(id int)
RETURNS TABLE (account_number char(8), sort_code char(6), type varchar(100), loan int,
balance int, name varchar(255), iban varchar(34), open_date date) AS $$
BEGIN
  -- returns bank account details
  RETURN QUERY
  SELECT a.account_number, a.account_sortcode, at.type_name, a.account_loan,
  a.account_balance, a.account_name, a.account_iban, a.account_opendate

  FROM customer.account a
  JOIN customer.account_type at ON a.account_type = at.type_id
  WHERE a.account_customerid = id;

-- if the customer does not exist
    IF NOT EXISTS (SELECT 1 FROM customer.customer WHERE customer_id=id) THEN
        RAISE EXCEPTION 'Customer does not exist';
    END IF;

END;
$$ LANGUAGE plpgsql;
```

Figure 5 - Function for checking bank account details

```sql
-- customer can see their balance function
CREATE OR REPLACE FUNCTION customer.check_balances(id int)
RETURNS TABLE (account_number char(8), account_name varchar(255),
account_type varchar(100), account_balance int) AS $$
BEGIN
  -- check if the customer exists
  RETURN QUERY
  SELECT a.account_number, a.account_name, at.type_name, a.account_balance
  FROM customer.account a
  JOIN customer.account_type at ON a.account_type = at.type_id
  WHERE a.account_customerid = id;

-- if the customer does not exist
    IF NOT EXISTS (SELECT 1 FROM customer.customer WHERE customer_id=id) THEN
        RAISE EXCEPTION 'Customer does not exist';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Figure 6 - Function for checking all balances of bank accounts

```
-- check transfers
CREATE OR REPLACE FUNCTION customer.check_transfers(IN c_id int)
RETURNS TABLE(transaction_id int, transaction_ref varchar(255), transfer_id int,
account_number char(8), receiver_accnum char(8), amount int, status varchar(50), date DATE) AS $$
#variable_conflict use_column
BEGIN
    RETURN QUERY
    SELECT t.pending_transactionid, t.pending_transactionref, t.pending_transferid, tr.transfer_senderaccnum,
    tr.transfer_receiveraccnum, tr.transfer_amount, tr.transfer_status, tr.transfer_date

    FROM customer.transaction_pending t
    JOIN customer.transfer tr ON t.pending_transferid = tr.transfer_id
    WHERE tr.transfer_senderaccnum IN (SELECT account_number FROM customer.account WHERE account_customerid = c_id);
END;
$$ LANGUAGE plpgsql;
```

Figure 7 - Function for checking transfers of a customer

```
-- check payments
CREATE OR REPLACE FUNCTION customer.check_payments(IN c_id int)
RETURNS TABLE(transaction_id int, transaction_ref varchar(255), payment_id int, account_number char(8),
receiver_accnum char(8), receiver_sortcode char(6), payment_receivername varchar(255), amount int,
status varchar(50), date DATE) AS $$
#variable_conflict use_column
BEGIN
    RETURN QUERY
    SELECT t.pending_transactionid, t.pending_transactionref, t.pending_paymentid, p.payment_accountnum,
    p.payment_receiveraccnum, p.payment_receiversortcode, p.payment_receivername, p.payment_amount,
    p.payment_status, p.payment_date

    FROM customer.transaction_pending t
    JOIN customer.payment p ON t.pending_paymentid = p.payment_id
    WHERE p.payment_accountnum IN (SELECT account_number FROM customer.account WHERE account_customerid = c_id);
END;
$$ LANGUAGE plpgsql;
```

Figure 8 - Function for checking payments of a customer

```sql
-- check loan
CREATE OR REPLACE FUNCTION customer.check_loans(IN c_id int)
RETURNS TABLE(transaction int, loan_id int, account_number char(8), amount int,
outstanding_balance int, interest real, term int, status varchar(50) ,date date) AS $$
BEGIN
    RETURN QUERY
    SELECT tp.pending_transactionid, l.loan_id, a.account_number, lt.loantype_amount,
    l.loan_outstanding, lt.loantype_interest, lt.loantype_term, l.loan_status ,l.loan_date

    FROM customer.account a
    JOIN customer.loan l ON a.account_loan = l.loan_id
    JOIN customer.loan_type lt ON l.loan_type = lt.loantype_id
    JOIN customer.transaction_pending tp ON l.loan_id = tp.pending_loanid
    WHERE a.account_customerid = c_id;

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE EXCEPTION 'No loans found';
END;
$$ LANGUAGE plpgsql;
```

Figure 9 - Function for checking all loans of a customer

```sql
-- create a new employee
CREATE OR REPLACE FUNCTION bank.create_employee(param_sortcode char(6), param_uname varchar(255),
param_pass varchar(255), param_role int, param_fname varchar(50), param_lname varchar(50),
param_mobile char(11), param_email varchar(50), param_address varchar(255), param_postcode varchar(10))

RETURNS TABLE(branch varchar(50), role varchar(255), first_name varchar(50), last_name varchar(50),
mobile char(11), email varchar(50), address varchar(255), postcode varchar(10)) AS $$
BEGIN
    INSERT INTO employee.employee (employee_id, employee_sortcode, employee_username, employee_password,
    employee_role, employee_fname, employee_lname, employee_mobile, employee_email, employee_address,
    employee_postcode)
    VALUES(nextval('employee.employee_employee_id_seq'), param_sortcode, param_uname, param_pass,
    param_role, param_fname, param_lname, param_mobile, param_email, param_address, param_postcode);

    RETURN QUERY
    SELECT br.branch_name, er.employeerole_name, e.employee_fname, e.employee_lname, e.employee_mobile,
    e.employee_email, e.employee_address, e.employee_postcode

    FROM employee.employee e
    JOIN employee.employee_roles er ON e.employee_role = er.employeerole_id
    JOIN bank.branch br ON e.employee_sortcode = br.branch_sortcode
    WHERE e.employee_id = currval('employee.employee_employee_id_seq');

END;
$$ LANGUAGE plpgsql;
```

Figure 10 - Function for creating a new employee at a specific branch

```
-- checking an employee details
CREATE OR REPLACE FUNCTION employee.check_employee(id int)
RETURNS TABLE (branch varchar(50), role varchar(255), first_name varchar(50), last_name varchar(50),
mobile char(11), email varchar(50), address varchar(255), postcode varchar(10)) AS $$
BEGIN
  -- returns employee details
  RETURN QUERY
  SELECT br.branch_name, er.employeerole_name, e.employee_fname, e.employee_lname, e.employee_mobile,
  e.employee_email, e.employee_address, e.employee_postcode

  FROM employee.employee e
  JOIN employee.employee_roles er ON e.employee_role = er.employeerole_id
  JOIN bank.branch br ON e.employee_sortcode = br.branch_sortcode
  WHERE e.employee_id = id;

-- if the employee does not exist
    IF NOT EXISTS (SELECT 1 FROM employee.employee WHERE employee_id=id) THEN
        RAISE EXCEPTION 'Employee does not exist';
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Figure 11 - Function for checking employee details

```sql
-- apply for loan
CREATE OR REPLACE FUNCTION bank.apply_loan(
    param_accountnum char(8),
    param_loantype int
)
RETURNS TABLE (id_loan int, type_loan int, status_loan varchar(50), date_loan date,
id_loantype int, amount_loantype int, interest_loantype real, term_loantype int ) AS $$
BEGIN
    IF EXISTS (SELECT 1 FROM customer.account WHERE account_number = param_accountnum) THEN
        -- insert transfer into customer.loan table
        INSERT INTO customer.loan (loan_id, loan_type, loan_status, loan_date)
        VALUES (nextval('customer.loan_loan_id_seq'), param_loantype, 'PENDING', NOW());

        -- update customer.account table with loan
        UPDATE customer.account
        SET account_loan = currval('customer.loan_loan_id_seq')
        WHERE account_number = param_accountnum;

        -- insert transfer into customer.transaction_pending table
        INSERT INTO customer.transaction_pending (pending_transactionid, pending_transactionref,
        pending_sensitiveflag, pending_approvalflag, pending_loanid)

        VALUES (nextval('customer.transaction_pending_pending_transactionid_seq'), 'LOAN', true,
        false, currval('customer.loan_loan_id_seq'));

    ELSE
        RAISE EXCEPTION 'Account does not exist';
    END IF;

    RETURN QUERY
    SELECT l.loan_id, l.loan_type, l.loan_status, l.loan_date, lt.loantype_id, lt.loantype_amount,
    lt.loantype_interest, lt.loantype_term

    FROM customer.loan l
    JOIN customer.loan_type lt ON l.loan_type = lt.loantype_id
    WHERE loan_id = currval('customer.loan_loan_id_seq');
END;
$$ LANGUAGE plpgsql;
```

Figure 12 - Function for applying for a loan

```
-- make a payment
CREATE OR REPLACE FUNCTION bank.make_payment(sender_accnum char(8), receiver_accnum char(8),
receiver_sortcode char(6), receiver_name varchar(255), amount int)

RETURNS TABLE (pay_id int, pay_account char(8), pay_receiveracc char(8), pay_receiversort char(6),
pay_receivername varchar(255), pay_amount int, pay_status varchar(50), pay_date DATE) AS $$

BEGIN
    IF amount <= 0 THEN
        RAISE EXCEPTION 'Cannot pay negative amount of money';
    END IF;

    -- check if sender account exists
    IF NOT EXISTS (SELECT 1 FROM customer.account WHERE account_number = sender_accnum) THEN
        RAISE EXCEPTION 'Sender account does not exist';
    END IF;

    -- check if sender has sufficient funds
    IF (SELECT account_balance FROM customer.account WHERE account_number = sender_accnum) < amount THEN
        RAISE EXCEPTION 'Insufficient funds in sender account';
    END IF;

    -- update sender account balance
    UPDATE customer.account SET account_balance = account_balance - amount WHERE account_number = sender_accnum;

    -- insert payment into customer.payment table
    INSERT INTO customer.payment (payment_id, payment_accountnum, payment_receiveraccnum, payment_receiversortcode,
    payment_receivername, payment_amount, payment_status, payment_date)

    VALUES (nextval('customer.payment_payment_id_seq'), sender_accnum, receiver_accnum, receiver_sortcode,
    receiver_name, amount, 'COMPLETE', NOW());

    -- insert payment into customer.transaction_pending table
    INSERT INTO customer.transaction_pending (pending_transactionid, pending_transactionref, pending_sensitiveflag,
    pending_approvalflag, pending_paymentid)
    VALUES (nextval('customer.transaction_pending_pending_transactionid_seq'), 'PAYMENT', false, true,
    currval('customer.payment_payment_id_seq'));

    INSERT INTO customer.transaction(transaction_id, transaction_complete)
    VALUES(nextval('customer.transaction_transaction_id_seq'),
    currval('customer.transaction_pending_pending_transactionid_seq'));

    RETURN QUERY
    SELECT *
    FROM customer.payment p
    WHERE p.payment_id = currval('customer.payment_payment_id_seq');

END;
$$ LANGUAGE plpgsql;
```

Figure 13 - Function for making a payment towards an external account

```sql
-- make a loan payment
CREATE OR REPLACE FUNCTION bank.pay_loan(param_accnum char(8), param_amount int)
RETURNS TABLE (pay_id int, pay_account char(8), pay_receiveracc char(8), pay_receiversort char(6),
pay_receivername varchar(255), pay_amount int, pay_status varchar(50), pay_date DATE) AS $$

BEGIN
    IF param_amount <= 0 THEN
        RAISE EXCEPTION 'Cannot pay negative amount of money.';
    END IF;
    -- check if sender account exists
    IF NOT EXISTS (SELECT 1 FROM customer.account WHERE account_number = param_accnum) THEN
        RAISE EXCEPTION 'Account does not exist.';
    END IF;
    -- check if sender has sufficient funds
    IF (SELECT account_balance FROM customer.account WHERE account_number = param_accnum) < param_amount THEN
        RAISE EXCEPTION 'Insufficient funds in account.';
    END IF;
    -- check if sender has sufficient funds
    IF (SELECT loan_outstanding FROM customer.loan l JOIN customer.account a ON l.loan_id = a.account_loan
    WHERE account_number = param_accnum) < param_amount THEN
        RAISE EXCEPTION 'You cannot pay more than the outstanding balance.';
    END IF;
    -- check if sender has a loan to pay back
    IF EXISTS (SELECT 1 FROM customer.account WHERE account_number = param_accnum AND account_loan IS NULL) THEN
        RAISE EXCEPTION 'No loan to pay back.';
    END IF;
    -- check if the loan is pending
    IF (SELECT loan_status FROM customer.loan l JOIN customer.account a ON l.loan_id = a.account_loan
    WHERE account_number = param_accnum) = 'PENDING' THEN
        RAISE EXCEPTION 'Loan is pending. You cannot pay back now.';
    END IF;
    -- update account balance
    UPDATE customer.account SET account_balance = account_balance - param_amount WHERE account_number = param_accnum;
    -- update loan outstanding balance
    UPDATE customer.loan l SET loan_outstanding = loan_outstanding - param_amount FROM customer.account a
    WHERE l.loan_id = a.account_loan AND a.account_number = param_accnum;

    -- insert payment into customer.payment table
    INSERT INTO customer.payment (payment_id, payment_accountnum, payment_receiveraccnum, payment_receiversortcode,
    payment_receivername, payment_amount, payment_status, payment_date)
    VALUES (nextval('customer.payment_payment_id_seq'), param_accnum, '00000000', '000000', 'LOAN PAYMENT',
    param_amount, 'COMPLETE', NOW());
    -- insert payment into customer.transaction_pending table
    INSERT INTO customer.transaction_pending (pending_transactionid, pending_transactionref, pending_sensitiveflag,
    pending_approvalflag, pending_paymentid)

    VALUES (nextval('customer.transaction_pending_pending_transactionid_seq'), 'LOAN PAYMENT', false, true,
     currval('customer.payment_payment_id_seq'));

    INSERT INTO customer.transaction(transaction_id, transaction_complete)
    VALUES(nextval('customer.transaction_transaction_id_seq'), currval('customer.transaction_pending_pending_transactionid_seq'));
    RETURN QUERY
    SELECT *
    FROM customer.payment p
    WHERE p.payment_id = currval('customer.payment_payment_id_seq');

END;
$$ LANGUAGE plpgsql;
```

Figure 14 - Function for paying back a loan

```
-- make a transfer
CREATE OR REPLACE FUNCTION bank.make_transfer(sender_accnum char(8), receiver_accnum char(8), amount int)
RETURNS TABLE (tran_id int, tran_senderacc char(8), tran_receiveracc char(8), tran_amount int,
tran_status varchar(50), tran_date date) AS $$
BEGIN
    IF ((SELECT account_customerid FROM customer.account WHERE account_number = sender_accnum) =
    (SELECT account_customerid FROM customer.account WHERE account_number = receiver_accnum)) THEN
        IF amount <= 0 THEN
            RAISE EXCEPTION 'Cannot transfer negative amount of money.';
        END IF;

        -- check if sender account exists
        IF NOT EXISTS (SELECT 1 FROM customer.account WHERE account_number = sender_accnum) THEN
            RAISE EXCEPTION 'Sender account does not exist.';
        END IF;

        -- check if receiver account exists
        IF NOT EXISTS (SELECT 1 FROM customer.account WHERE account_number = receiver_accnum) THEN
            RAISE EXCEPTION 'Receiver account does not exist.';
        END IF;

        -- check if sender has sufficient funds
        IF (SELECT account_balance FROM customer.account WHERE account_number = sender_accnum) < amount THEN
            RAISE EXCEPTION 'Insufficient funds in sender account.';
        END IF;

        -- update sender account balance
        UPDATE customer.account SET account_balance = account_balance - amount WHERE account_number = sender_accnum;

        -- update receiver account balance
        UPDATE customer.account SET account_balance = account_balance + amount WHERE account_number = receiver_accnum;

        -- insert transfer into customer.transfer table
        INSERT INTO customer.transfer (transfer_id, transfer_senderaccnum, transfer_receiveraccnum, transfer_amount,
        transfer_status, transfer_date)
        VALUES (nextval('customer.transfer_transfer_id_seq'), sender_accnum, receiver_accnum, amount, 'COMPLETE', NOW());

        -- insert transfer into customer.transaction_pending table
        INSERT INTO customer.transaction_pending (pending_transactionid, pending_transactionref, pending_sensitiveflag,
        pending_approvalflag, pending_transferid)
        VALUES (nextval('customer.transaction_pending_pending_transactionid_seq'), 'TRANSFER', false, true,
        currval('customer.transfer_transfer_id_seq'));

        INSERT INTO customer.transaction(transaction_id, transaction_complete)
        VALUES(nextval('customer.transaction_transaction_id_seq'), currval('customer.transaction_pending_pending_transactionid_seq'));

    ELSE
        RAISE EXCEPTION 'The receiving account does not exist or is not yours. You can only transfer money between your own accounts.';
    END IF;

    RETURN QUERY
    SELECT *
    FROM customer.transfer t
    WHERE t.transfer_id = currval('customer.transfer_transfer_id_seq');
END;
$$ LANGUAGE plpgsql;
```

Figure 15 - Function for making a transfer

```
-- manager approval of transaction procedure (loans, credit limits)
CREATE OR REPLACE FUNCTION manager.approve_pending(param_employee int, tran_id int)
RETURNS TABLE(appr_id int, appr_employee int, appr_date date, appr_tran int) AS $$
BEGIN
    IF (SELECT er.employeerole_name FROM employee.employee e JOIN employee.employee_roles er
    ON e.employee_role = er.employeerole_id WHERE e.employee_id = param_employee) = 'Manager' THEN
        -- Update pending transactions with sensitive flag set to true if sensitive flag is set to true
        IF EXISTS (SELECT * FROM customer.transaction_pending WHERE pending_sensitiveflag = true
        AND pending_approvalflag = false AND pending_transactionid = tran_id) THEN

            UPDATE customer.transaction_pending
            SET pending_approvalflag = true
            WHERE pending_sensitiveflag = true AND pending_transactionid = tran_id;

            IF EXISTS (SELECT pending_transferid FROM customer.transaction_pending
            WHERE pending_approvalflag = true AND pending_transactionid = tran_id) THEN
                UPDATE customer.transfer t
                SET transfer_status = 'COMPLETE'
                FROM customer.transaction_pending tr
                WHERE t.transfer_id = tr.pending_transferid;
            END IF;

            IF EXISTS (SELECT pending_paymentid FROM customer.transaction_pending
            WHERE pending_approvalflag = true AND pending_transactionid = tran_id) THEN
                UPDATE customer.payment p
                SET payment_status = 'COMPLETE'
                FROM customer.transaction_pending tr
                WHERE p.payment_id = tr.pending_paymentid;
            END IF;

            IF EXISTS (SELECT pending_loanid FROM customer.transaction_pending
            WHERE pending_approvalflag = true AND pending_transactionid = tran_id) THEN
                UPDATE customer.loan l
                SET loan_status = 'ACTIVE'
                FROM customer.transaction_pending tr
                WHERE l.loan_id = tr.pending_loanid;

                UPDATE customer.loan
                SET loan_outstanding =
                (
                    SELECT lt.loantype_amount
                    FROM customer.transaction_pending tp
                    JOIN customer.loan l ON l.loan_id = tp.pending_loanid
                    JOIN customer.loan_type lt ON l.loan_type = lt.loantype_id
                    WHERE tp.pending_transactionid = tran_id
                )
                * (( SELECT lt.loantype_interest
                    FROM customer.transaction_pending tp
                    JOIN customer.loan l ON l.loan_id = tp.pending_loanid
                    JOIN customer.loan_type lt ON l.loan_type = lt.loantype_id
                    WHERE tp.pending_transactionid = tran_id
                )+1)

                FROM customer.transaction_pending tr
                WHERE loan_id = tr.pending_loanid;
```

Figure 16 - Function for manager approval of pending transaction

```
            UPDATE customer.account
            SET account_balance = account_balance + (SELECT loantype_amount FROM customer.loan_type lt
            JOIN customer.loan l ON l.loan_type = lt.loantype_id WHERE l.loan_id = (SELECT pending_loanid
            FROM customer.transaction_pending WHERE pending_approvalflag = true AND pending_transactionid = tran_id))

            WHERE account_loan = (SELECT pending_loanid FROM customer.transaction_pending
            WHERE pending_approvalflag = true AND pending_transactionid = tran_id);

        END IF;

        -- Update approvals record
        INSERT INTO manager.approval(approval_id, approval_employee, approval_date,approval_transaction)
        VALUES (nextval('manager.approval_approval_id_seq'), param_employee, NOW(), tran_id);

        -- Update transaction record
        INSERT INTO customer.transaction(transaction_id, transaction_complete)
        VALUES(nextval('customer.transaction_transaction_id_seq'), tran_id);

    ELSIF EXISTS (SELECT * FROM customer.transaction_pending WHERE (pending_sensitiveflag = false
    OR pending_approvalflag = true) AND pending_transactionid = tran_id) THEN
        RAISE EXCEPTION 'Transaction is not sensitive or does not need approval.';

    ELSE
        RAISE EXCEPTION 'No transaction found';

    END IF;

ELSE
    RAISE EXCEPTION 'Employee is not a manager';
END IF;

RETURN QUERY
SELECT *
FROM manager.approval a
WHERE a.approval_id = currval('manager.approval_approval_id_seq');
END;
$$ LANGUAGE plpgsql;
```

Figure 17 - Function for manager approval of pending transaction cont.

```
--creation of roles and users

-- bank role and users, permission and privileges
CREATE ROLE role_bank;
CREATE ROLE user_bank1 WITH LOGIN PASSWORD 'test';

GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA bank TO role_bank;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA manager TO role_bank;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA employee TO role_bank;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA customer TO role_bank;

GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA customer TO role_bank;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA employee TO role_bank;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA manager TO role_bank;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA bank TO role_bank;

GRANT USAGE ON SCHEMA bank TO role_bank;
GRANT USAGE ON SCHEMA manager TO role_bank;
GRANT USAGE ON SCHEMA employee TO role_bank;
GRANT USAGE ON SCHEMA customer TO role_bank;

GRANT role_bank TO user_bank1;
```

Figure 18 - Permissions and creation of bank role and user

```
-- customer role and users, permission and privileges
CREATE ROLE role_customer;
CREATE ROLE user_customer1 WITH LOGIN PASSWORD 'test';

GRANT USAGE ON SCHEMA customer TO role_customer;

GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA customer TO role_customer;
GRANT SELECT ON ALL TABLES IN SCHEMA customer TO role_customer;

GRANT role_customer TO user_customer1;
```

Figure 19 - Permissions and creation of customer role and user

```
-- employee role and users, permission and privileges
CREATE ROLE role_employee;
CREATE ROLE user_employee1 WITH LOGIN PASSWORD 'test';

GRANT USAGE ON SCHEMA employee TO role_employee;
GRANT SELECT ON ALL TABLES IN SCHEMA employee TO role_employee;

GRANT USAGE ON SCHEMA bank TO role_employee;
GRANT SELECT ON TABLE bank.branch TO role_employee;

GRANT role_employee TO user_employee1;
GRANT role_customer TO user_employee1;
```

Figure 20 - Permissions and creation of employee role and user

```
-- manager role and users, permission and privileges
CREATE ROLE role_manager;
CREATE ROLE user_manager1 WITH LOGIN PASSWORD 'test';

GRANT USAGE ON SCHEMA manager TO role_manager;
GRANT SELECT, UPDATE, INSERT ON ALL TABLES IN SCHEMA manager TO role_manager;
GRANT USAGE, SELECT ON ALL SEQUENCES IN SCHEMA manager TO role_manager;

GRANT UPDATE ON TABLE customer.transaction_pending TO role_manager;
GRANT INSERT ON TABLE customer.transaction TO role_manager;
GRANT UPDATE ON TABLE customer.transfer TO role_manager;
GRANT UPDATE ON TABLE customer.payment TO role_manager;
GRANT UPDATE ON TABLE customer.account TO role_manager;
GRANT SELECT ON TABLE customer.loan_type TO role_manager;
GRANT SELECT, UPDATE ON TABLE customer.loan TO role_manager;

GRANT USAGE, SELECT ON SEQUENCE customer.transaction_transaction_id_seq TO role_manager;

GRANT role_manager TO user_manager1;
GRANT role_employee TO user_manager1;
GRANT role_customer TO user_manager1;
```

Figure 21 - Permissions and creation of manager role and user

```
#!/bin/bash

sudo -su postgres psql -U postgres -c "ALTER ROLE postgres WITH LOGIN PASSWORD 'postgres';"

export PGHOST=localhost
export PGPORT=5433
export PGUSER=postgres
export PGPASSWORD=postgres

psql -c "DROP DATABASE IF EXISTS banking";
psql -c "CREATE DATABASE banking";
psql -U postgres -d banking < banking.sql;

export PGDATABASE=banking

psql -c "INSERT INTO bank.bank (bank_id, bank_name, bank_bic, bank_address, bank_postcode, bank_country) VALUES
(nextval('bank.bank_bank_id_seq'),'LLOYDS PANK PLC','LOYDGB2LXXX','25 MONUMENT STREET','EC3R8BQ','United Kingdom');
"


psql -c "INSERT INTO bank.branch (branch_sortcode, branch_bankid, branch_name, branch_address, branch_postcode, branch_country) VALUES
('309921',1,'Lloyds Bank Watford','Po Box 1000','BX11LT','United Kingdom'),
('386650',1,'Lloyds Bank Warwick','Po Box WARWICK','CV354LT','United Kingdom');
"


psql -c "INSERT INTO customer.account_type (type_id, type_name) VALUES
(nextval('customer.account_type_type_id_seq'),'Current Account');
"


psql -c "INSERT INTO customer.loan_type (loantype_id, loantype_amount, loantype_interest, loantype_term) VALUES
(nextval('customer.loan_type_loantype_id_seq'),10000,0.054,60),
(nextval('customer.loan_type_loantype_id_seq'),25000,0.069,60);
"


psql -c "INSERT INTO customer.loan (loan_id, loan_type, loan_status, loan_date) VALUES
(nextval('customer.loan_loan_id_seq'),1,'PENDING', NOW());
"


psql -c "INSERT INTO customer.customer (customer_id, customer_username, customer_password, customer_fname, customer_lname, customer_mobile,
customer_email, customer_address, customer_postcode) VALUES
(nextval('customer.customer_customer_id_seq'),'aferguson1','testpass','Andrew','Ferguson','07941083085','andrewferguson@gmail.com',
'10 Abbey Close, Coventry','CV56HN'),
(nextval('customer.customer_customer_id_seq'),'mpearlo1','passtest','Maria','Pearlo','07458634123','mariapearlo123@gmail.com',
'156 Lowley Road, Warwick','CV344XP');
"


psql -c "INSERT INTO customer.account (account_number, account_sortcode, account_customerid, account_type, account_loan, account_balance,
account_name, account_iban, account_opendate) VALUES
('68932868','309921',1,1,NULL,2000,'bank account 1 andrew','GB73LOYD30992168932868',NOW()),
('68932777','309921',1,1,NULL,500,'bank account 2 andrew','GB73LOYD30992168932777',NOW()),
('68156222','386650',2,1,NULL,10000,'bank account 1 maria','GB73LOYD38665068156222',NOW()),
('68156345','386650',2,1,1,65000,'bank account 2 maria','GB73LOYD38665068156345',NOW());
"
```

Figure 22 - Inserting test data from the test script

```
psql -c "INSERT INTO customer.payment (payment_id, payment_accountnum, payment_receiveraccnum, payment_receiversortcode,
payment_receivername, payment_amount, payment_status, payment_date) VALUES
(nextval('customer.payment_payment_id_seq'),'68932777','12345678','112233','John Ferguson',500,'COMPLETE',NOW()),
(nextval('customer.payment_payment_id_seq'),'68156222','65423133','112233','Johnny Pearlo',500,'PENDING',NOW()),
(nextval('customer.payment_payment_id_seq'),'68156345','98732123','112233','Janay Pearlo',500,'COMPLETE',NOW()),
(nextval('customer.payment_payment_id_seq'),'68932868','87654321','332211','Jane Ferguson',220,'PENDING',NOW());
"

psql -c "INSERT INTO customer.transfer (transfer_id, transfer_senderaccnum, transfer_receiveraccnum, transfer_amount, transfer_status,
transfer_date) VALUES
(nextval('customer.transfer_transfer_id_seq'),'68932868','68932777',300,'COMPLETE',NOW()),
(nextval('customer.transfer_transfer_id_seq'),'68932777','68932868',300,'PENDING',NOW()),
(nextval('customer.transfer_transfer_id_seq'),'68156222','68156345',1200,'PENDING',NOW());
"

psql -c "INSERT INTO customer.transaction_pending (pending_transactionid, pending_transactionref, pending_transferid,
pending_paymentid, pending_loanid, pending_sensitiveflag, pending_approvalflag) VALUES
(nextval('customer.transaction_pending_pending_transactionid_seq'),'PAYMENT',NULL,1,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'PAYMENT',NULL,2,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'PAYMENT',NULL,3,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'PAYMENT',NULL,4,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'TRANSFER',1,NULL,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'TRANSFER',2,NULL,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'TRANSFER',3,NULL,NULL,false,true),
(nextval('customer.transaction_pending_pending_transactionid_seq'),'LOAN',NULL,NULL,1,true,false);
"

psql -c "INSERT INTO customer.transaction (transaction_id, transaction_complete) VALUES
(nextval('customer.transaction_transaction_id_seq'),1),
(nextval('customer.transaction_transaction_id_seq'),2),
(nextval('customer.transaction_transaction_id_seq'),3),
(nextval('customer.transaction_transaction_id_seq'),4),
(nextval('customer.transaction_transaction_id_seq'),5),
(nextval('customer.transaction_transaction_id_seq'),6),
(nextval('customer.transaction_transaction_id_seq'),7);
"


# psql -c "INSERT INTO manager.approval (approval_id, approval_employee, approval_date, approval_flag) VALUES
# ();
# "

psql -c "INSERT INTO employee.employee_roles (employeerole_id, employeerole_name) VALUES
(nextval('employee.employee_roles_employeerole_id_seq'),'Teller'),
(nextval('employee.employee_roles_employeerole_id_seq'),'Manager');
"

psql -c "INSERT INTO employee.employee (employee_id, employee_sortcode, employee_username, employee_password, employee_role, employee_fname,
employee_lname, employee_mobile, employee_email, employee_address, employee_postcode) VALUES
(nextval('employee.employee_employee_id_seq'),'309921','employeeuser','employeepass',1,'John','Doe','07495381704','johndoe@gmail.com',
'321 Regent Street','N127JE'),
(nextval('employee.employee_employee_id_seq'),'309921','manageruser','managerpass',2,'Jane','Doe','07495385874','janedoe@gmail.com',
'1 Market Street','W36PE');
"
```

Figure 23 - Inserting test data from the test script cont.

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ starting to test all procedures @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_bank1
export PGPASSWORD=test

# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ customer creation journey @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system user journey for back end
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system creating a customer
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.create_customer
('mwarren','password123','Michael','Warren','07942173055',
'mwarren@gmail.com','109 Durban Rd W, Watford', 'WD187DT');"


echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system creating an account
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.create_account
('mwarren', '68324767','309921',1,'my first personal bank account');"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system creating another account
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.create_account
('mwarren', '68324676','309921',1,'my second personal bank account');"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system applying for a loan with customer account number
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.apply_loan('68324767',1)"
```

Figure 24 - Testing the customer creation journey and applying for a loan

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ employee and manager creation journey @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
creating employees with the back end
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system creating an employee
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.create_employee('309921','abancu','password123',1,
'Andreea','Bancu','07495172653','abancu@gmail.com','14 Granville Rd, Watford',' WD180AH')"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system creating a manager
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.create_employee('309921','msmith','password123',2,
'Maria','Smith','07435225881','msmith@gmail.com','122 Anderson Rd, Watford',' WD225AD')"
```

Figure 25 - Testing the employee and manager creation journey

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ employee checking a customer's balances @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_employee1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
employee checking a customer's balances
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer 1
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(1)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer 2
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(2)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer 3
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(3)"
```

Figure 26 - Testing an employee checking a customer's balance

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@ manager approving pending transactions @@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_manager1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@
manager checking loans
@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_loans(3)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@
manager approving loan
@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from manager.approve_pending(2,9)"
```

Figure 27 - Testing the manager approving pending transactions

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@ customer checking their accounts @@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_customer1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking if their loan was approved
@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking account balances
@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(3)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking loans
@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_loans(3)"
```

Figure 28 - Testing a customer checking their loans and balances

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ customer paying their loan back @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@


export PGUSER=user_bank1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer loan payments, payments and transfers
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer paying their loan back
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.pay_loan('68324767',500)"
```

Figure 29 - Testing a customer paying their loan back

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@ customer making a transfer between
# their own accounts then checking transfers and balances @@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_bank1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@
customer making a transfer between their own accounts
@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.make_transfer('68324767','68324676', 1000)"
psql -c "select * from bank.make_transfer('68324676','68324767', 200)"

export PGUSER=user_customer1
export PGPASSWORD=test


echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking transfers
@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_transfers(3)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking balances
@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(3)"
```

Figure 30 - Testing a customer making transfers then checking their transfers and balances

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@ testing erroneous data and unwanted schema access @@@@@@@@@@@@@@@@@@@@@@@@@@@@
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@ customer testing @@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_customer1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing customer role
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in bank schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.make_transfer('68324767','68324676', 1000)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in employee schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from employee.check_employee(2)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in manager schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from manager.approve_pending(2,9)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking wrong customer details
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_customer(7)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking wrong account details
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_accounts(7)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer checking wrong balance details
@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_balances(7)"
```

Figure 31 - Testing erroneous data and unauthorised access from a customer user

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ employee testing @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_employee1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing employee role
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
employee accessing functions in bank schema
(has access to bank schema as bank.branch_sortcode attribute is needed;
but does not have access to updating any tables)
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.make_transfer('68324767','68324676', 1000)"


echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
employee accessing functions in manager schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from manager.approve_pending(2,9)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
employee checking wrong employee details
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from employee.check_employee(7)"
```

Figure 32 - Testing erroneous data and unauthorised access from an employee user

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ manager testing @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_manager1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing manager role
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
manager accessing functions in bank schema
(has access to bank schema as bank.branch_sortcode attribute is needed;
but does not have access to updating any tables)
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.make_transfer('68324767','68324676', 1000)"


echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
manager uses wrong employee details to approve transaction
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from manager.approve_pending(1,8)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
manager uses non-existent transaction details to approve transaction
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from manager.approve_pending(2,21)"
```

Figure 33 - Testing erroneous data and unauthorised access from a manager user

```
# @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ bank testing @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

export PGUSER=user_bank1
export PGPASSWORD=test

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing bank
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system pays loan with negative money
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.pay_loan('68324767',-100000)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system pays loan with more money than available
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.pay_loan('68324767',100000)"

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system pays to the wrong account
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.pay_loan('12345678',11000)"

echo
echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing paying to a pending loan
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system applying for a loan with customer account number
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.apply_loan('68932777',2)"



echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank checking loans
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from customer.check_loans(1)"


echo && echo "
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
bank system tries to pay to a pending loan
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@"
echo

psql -c "select * from bank.pay_loan('68932777',50)"
```

Figure 34 - Testing erroneous data and unauthorised access from a bank user

```
# IPv4 local connections:
hostssl    postgres        postgres            127.0.0.1/32        scram-sha-256
hostssl    banking         postgres            127.0.0.1/32        scram-sha-256
hostssl    banking         user_bank1          127.0.0.1/32        scram-sha-256
hostssl    banking         user_manager1       127.0.0.1/32        scram-sha-256
hostssl    banking         user_employee1      127.0.0.1/32        scram-sha-256
hostssl    banking         user_customer1      127.0.0.1/32        scram-sha-256

# IPv6 local connections:
#host    all             all             ::1/128                 scram-sha-256
```

Figure 35 - Changes made to pg_hba.conf file

```
# connections
listen_addresses = '127.0.0.1'
port = 5433
max_connections = 100

# authentication and passwords
authentication_timeout = 30s
password_encryption = scram-sha-256

#ssl on
ssl = on
ssl_cert_file = '/etc/ssl/certs/ssl-cert-snakeoil.pem'
ssl_key_file = '/etc/ssl/private/ssl-cert-snakeoil.key'

# logs
log_destination = 'stderr'
logging_collector = on
log_directory = 'log'
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_file_mode = 0600


# tracking database activity, use SELECT * FROM pg_stat_activity; to see
track_activities = on
track_activity_query_size = 1024
track_counts = on
track_io_timing = on
track_wal_io_timing = on
track_functions = all
```

Figure 36 - Changes made to postgresql.conf file

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@
testing customer role
@@@@@@@@@@@@@@@@@@@@@@@@@@@@



@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in bank schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@

ERROR:  permission denied for schema bank
LINE 1: select * from bank.make_transfer('68324767','68324676', 1000...
                      ^


@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in employee schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@

ERROR:  permission denied for schema employee
LINE 1: select * from employee.check_employee(2)
                      ^


@@@@@@@@@@@@@@@@@@@@@@@@@@@@
customer accessing functions in manager schema
@@@@@@@@@@@@@@@@@@@@@@@@@@@@

ERROR:  permission denied for schema manager
LINE 1: select * from manager.approve_pending(2,9)
                      ^
```

Figure 37 - Customer trying to access unauthorised schemas