2023

An Online Banking System

INFORMATION MANAGEMENT

U2136249

Table of Contents

Summary
This report provides a comprehensive understanding of the technical aspects behind of an online banking system. It outlines the user flow and the various functionalities that support the system's service layer. To ensure compliance with GDPR, the design of the system's back-end incorporates best practices set by the Information Commissioner's office, including efficient data collection, processing, and storage.

Data Model
I started the data model by illustrating a high-level structure of the bank shown in Figure 1, which outlines all the entities (displayed as rectangles), and logic patterns (displayed as diamonds) of the bank. The user beings its journey by registering themselves on the platform. After completing the registration process, the customer is then able to open an online bank account, giving them access to features such as checking their account balance, applying for loans, and transferring funds between accounts.

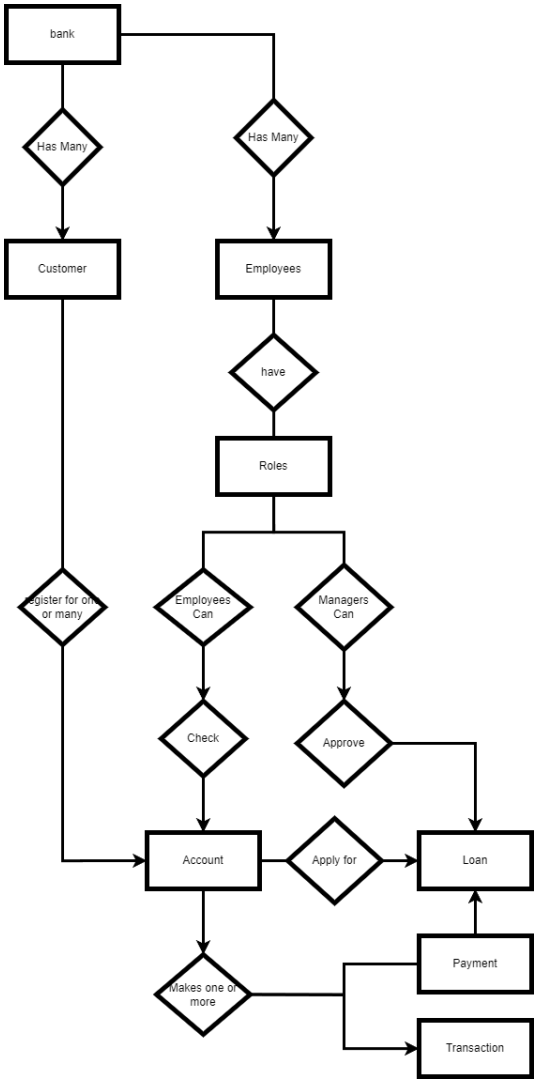

Figure 1 - Structure of the Online Banking System.

Entity Relationship Diagram (ERD)

After laying out the overall structure and logic of the online bank system, I then began to populate all the correspondent attributes for the entities needed to meet the specification in their normalised form, illustrated in Figure 2.

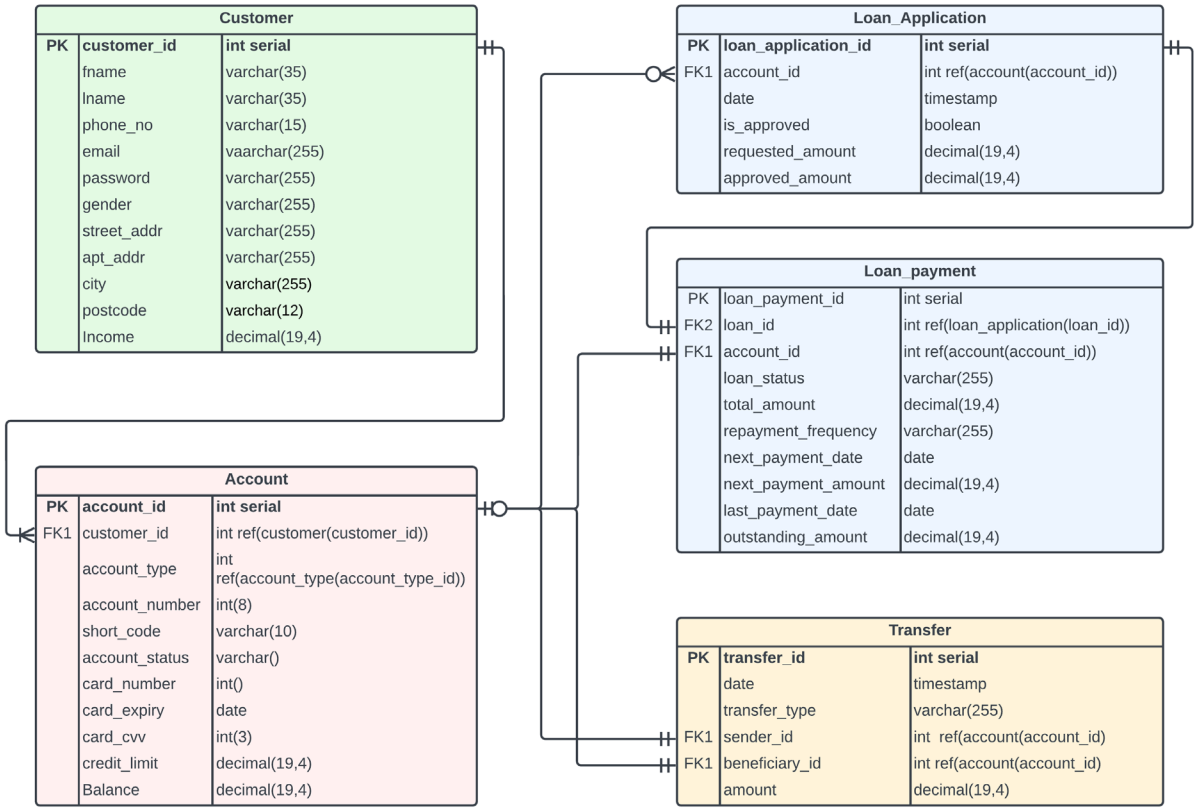

Figure 2 - Entity Relationship Diagram for an Online Bank System.

Figure 2 is an entity relationship diagram that shows the relationships between attributes across different entities to enable the structure flow described above to construct a sufficient data model for an online-bank system.

The relationships were constructed using crow's foot notation (Dybka, 2016), and are as follows:

1. Customers have a "one-to-many" relationship with the Account entity, where a customer can have one or many accounts, but an account can only belong to one customer.
2. Account has a "one-to-many" relationship with the Loan Application entity, where an account can have multiple loan applications, but a loan application can only belong to one account.
3. Loan Application has a "one-to-many" relationship with Loan Payment, where a loan application can have multiple loan payments, but a loan payment can only be made to one loan application.
4. Customer has a one-to-many relationship with Transfer, where a customer can have multiple transfers, but a transaction can only be from one customer.
5. Transfer has a "many-to-one" relationship with Account, where multiple transactions can be associated with one account, but an account can only be associated with one transaction at a time.

Implementation of Entities and Attributes

Following the construction of the EDR diagram, I began to implement of the data model using PostgreSQL language. I started by identifying the data type and field sizes of all entity attributes corresponding conditions.

Normalised Customer Entity

Attributes	***Data Type***	***Conditions***
Id (PK)	INT	SERIAL
Fname	VARCHAR(35)	NOT NULL
Lname	VARCHAR(35)	NOT NULL
Phone_Number	VARCHAR(15)	NOT NULL
Email	VARCHAR(255)	NOT NULL
Password	VARCHAR(255)	NOT NULL
Gender	VARCHAR(1)	NOT NULL
Date_of_birth	DATE	NOT NULL
Street_addr	VARCHAR(255)	NOT NULL
House_addr	VARCHAR(255)	NOT NULL
City	VARCHAR(255)	NOT NULL
Postcode	VARCHAR(12)	NOT NULL
Employment_status	VARCHAR(10)	CONSTAINT
Income	DECIMAL(19,4)	MONEY

Table 1 – Normalised Customer Entity.

Customer Entity Implementation

```
-- Create Customer Table
CREATE TABLE customer (
    id INT GENERATED BY DEFAULT AS IDENTITY ,
    fname VARCHAR(35) NOT NULL,
    lname VARCHAR(35) NOT NULL,
    phone_number VARCHAR(15) NOT NULL,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    gender VARCHAR(1) NOT NULL,
    date_of_birth DATE NOT NULL,
    street_addr VARCHAR(255) NOT NULL,
    house_addr VARCHAR(255),
    city VARCHAR(255) NOT NULL,
    postcode VARCHAR(12) NOT NULL,
    employment_status VARCHAR(10)
    income decimal(19,4) NOT NULL,
    PRIMARY KEY (id),
    CONSTRAINT gender_check CHECK (gender = ANY (ARRAY['F'::bpchar, 'M'::bpchar])), --
    check user input, only valid 'F' for Female and 'M' for Male.
    CONSTRAINT employment_check CHECK (employment_check = ANY (ARRAY['Employed'::bpchar,
    'Unemployed'::bpchar]))
);
```

Figure 3 - PostgreSQL Implementation of Customer Entity.

The customer entity is the entry point for customers, it allows them to register for a new online bank account. Table 1, shows all the attributes needed for a successful registration, which have been normalised to its maximum form to reduce data redundancy and improve data integrity (Microsoft, 2022). For example, the customer's full name has been broken down into "fname" (first name) and "lname" (last name) and their field size to be 35 bytes. I have done this to ensure compliance with

data standards set out by governmental institutions (Cabinet Office, 2021), which this is to maximise database storage optimisation.

Figure 3, shows the implementation code using the PostgreSQL language. I first created a table named "customer" containing the attributes shown in Table 1. To uniquely identify a customer record, I set a primary key (PK) as "ID" which is generated by the database automatically as new customer are added to the table, this minimises data redundancy and allow for better system efficiency (IBM, 2023). For certain columns, such as "gender" and "employment_status", I set constraints to check and validate the user input. For example, "gender" can only be 'F' for female or 'M' for male.

Normalised Account Entity

Attributes	Data Type	Condition
Id (PK)	INT	SERIAL
Customer_id (FK)	INT	NOT NULL
Account_type	VARCHAR(7)	CONSTRAINT
Account_numer	INT	NOT NULL
Short_code	VARCHAR(10)	NOT NULL
Card_number	INT	NOT NULL
Card_expiry	DATE	NOT NULL
Credit_limit	DECIMAL(19,4)	MONEY
Balance	DECIMAL(19,4)	MONEY

Table 2 - Account Entity.

Account Entity Implementation

```
-- Create Account Table
CREATE TABLE account (
    id int GENERATED BY DEFAULT AS IDENTITY,
    customer_id INTEGER NOT NULL,
    account_type varchar(7) NOT NULL,
    account_number int NOT NULL,
    short_code varchar(10) NOT NULL,
    card_number int NOT NULL,
    card_expiry DATE NOT NULL,
    card_cvv int NOT NULL,
    credit_limit decimal(19,4) NOT NULL,
    balance decimal(19,4) NOT NULL DEFAULT 0,
    PRIMARY KEY (id),
    FOREIGN KEY (customer_id) REFERENCES customer(id) ON DELETE CASCADE,
    CONSTRAINT type_of_account CHECK (account_type = ANY (ARRAY['Savings'::bpchar,
    'Current'::bpchar, 'Credit'::bpchar]))
);
```

Figure 4 - PostgreSQL Implementation of Account Entity.

Figure 4, details implementation of the account entity. I started by creating a table called "account", defining all the normalised attributes outlined in Table 2 necessary to compose an online bank account. To start with I defined the primary key to be "ID" which will act as an identifies for other tables such as the transfer entity to identify where a transaction is going to. I then defined a foreign

key (FK) to provide a link between the “customer” and the “account” entities to provide an identification of who customer owns which account (Rabelo, 2019). I also set rules for some of the data in the table, such as “NOT NULL” which ensures that some type of data must be entered and must never be left blank, otherwise the system will raise an error. “Customer_id” attribute is set to be “ON DELETE CASCADE” which means that when a customer is deleted, all related accounts will be deleted as well. Finally, “account_type” has a “CHECK” constraint which is used to validate the user input to be either; savings, current or credit.

Loan Application Entity Overview

Attributes	Data Type	Condition
Id (PK)	INT	SERIAL
Account_id (FK)	INT	SERIAL
Date	TIMESTAMP	NOT NULL
Requested_amount	DECIMAL(19,4)	MONEY
Is_approved	BOOLEAN	NOT NULL
Approved_amount	DECIMAL(19,4)	MONEY

Table 3 - Loan Application Entity.

Loan Application Entity Implementation

```
-- Create Loan Application Table
CREATE TABLE loan_application (
    id int GENERATED BY DEFAULT AS IDENTITY,
    account_id INTEGER,
    date TIMESTAMP WITH TIME ZONE,
    requested_amount decimal(19,4),
    is_approved boolean NOT NULL,
    approved_amount decimal(19,4) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (account_id) REFERENCES account(id) ON DELETE CASCADE
);
```

Figure 5 - PostgreSQL Implementation of Loan Application Entity.

The purpose of the loan application table is to allow managers to see incoming applications made by existing customers. Figure 5, shows the implementation of all the normalised attributes shown in Table 3. The customer would begin the application process by requesting an specific amount and depending on whether the customer has a job, or their income is above a certain amount the manger then is able to make a decision on whether to “approve” the loan or not. The Boolean attribute “is_approved” acts as the determinator for the application to be approved (TRUE) or denied (FALSE). To follow GPDR guidance the “requested_amount” and the “approved_amount” are defined as “decimal(19,4)” which is the recommended data standard by the in Information Commission Office (ICO, 2022).

Loan Payment Entity Overview

Attributes	***Data Type***	***Condition***

Id (PK)	INT	SERIAL
date	TIMESTAMP	NOT NULL
Customer_id (FK)	INT	SERIAL
Transfer_type	VARCHAR	CONSTRAINT
Sender_id (FK)	INT	SERIAL
Receiver_id (FK)	INT	SERIAL
amount	DECIMAL(19,4)	MONEY

Table 4 - Loan Payment Entity.

Loan Payment Implementation

```
-- Create Loan Payment Table
CREATE TABLE loan_payment (
    id INT GENERATED BY DEFAULT AS IDENTITY,
    loan_application_id INTEGER,
    account_id INTEGER DEFAULT 0,
    loan_status varchar(8) NOT NULL,
    total_amount decimal(19,4) NOT NULL,
    repayment_frequency varchar(255),
    next_payment_date DATE,
    next_payment_amount decimal(19,4),
    last_payment_date DATE,
    outstanding_amount decimal(19,4) NOT NULL,
    PRIMARY KEY (id),
    FOREIGN KEY (loan_application_id) REFERENCES loan_application(id) ON DELETE CASCADE
);
```

Figure 6 - PostgreSQL Implementation of Loan Payment Entity.

Figure 7, details the code implementation of the “loan_payment” entity, which enables existing customers to make payments to approved loans. To start with, I created the table named “loan_payment” with the specified columns and constraints shown in Table 4. I then defined the “ID” as a primary key which will be generated by default to act as the identity column. The “loan_application_id” column is set as the FK with the “ON DELETE CASCADE” constraint, which ensures that if a loan application is deleted, all related loan payments will be deleted as well.

Transfer Overview

Attributes	***Data Type***	***Condition***
Id (PK)	INT	SERIAL
date	TIMESTAMP	NOT NULL
Customer_id (FK)	INT	SERIAL
Transfer_type	VARCHAR	CONSTRAINT
Sender_id (FK)	INT	SERIAL
Receiver_id (FK)	INT	SERIAL
amount	DECIMAL(19,4)	MONEY

Table 5 - Transfer Entity

Transfer Implementation

```
-- Create Transfer Table
CREATE TABLE transfer (
    id int GENERATED BY DEFAULT AS IDENTITY,
    date TIMESTAMP WITH TIME ZONE NOT NULL,
    customer_id INTEGER,
    transfer_type varchar(255) NOT NULL,
    sender_id INTEGER NOT NULL,
    receiver_id INTEGER NOT NULL,
    amount decimal(19,4) NOT NULL DEFAULT 0,
    PRIMARY KEY (id),
    FOREIGN KEY (customer_id) REFERENCES customer(id) ON DELETE CASCADE,
    FOREIGN KEY (sender_id) REFERENCES account(id),
    FOREIGN KEY (receiver_id) REFERENCES account(id),
    CONSTRAINT amount_check CHECK (amount > 0),
    CONSTRAINT valid_transfer UNIQUE (sender_id, receiver_id)
);
```

Figure 7 - PostgreSQL Implementation of Transfer Entity.

Figure 7, illustrates the implementation of the "transfer" entity containing all the attributes shown in Table 5. The purpose is to allow customer to make transfers to pay for loans, or transfer balance from one account to another.

It starts by defining its primary key "id", referencing the "customer" through "customer_id" as an FK. "Sender_id" and "receiver_id" both reference to the "account" table, to avoid the customer from sending money to the same account, I have implemented constraints that will ensure the two values are unique. I have also implemented a check in the "balance" attribute to make sure that the "sender's" balance is above 0 to avoid errors when making a transfer.

Business Logic

Roles

To ensure the maintenance of the database and fulfilment of the database requirements, I implemented database system roles. Roles allow for easy manager of user privileges as admins users can assign privileges to a role, rather than to individual users (postgreSQL, 2022).

For the online bank system, I implemented two database roles: "manager" and "customer". The "manager" role is granted all the privileges on all entities of the system, this includes on the; "customer", "account", "loan_application", "loan_payment" and "transfers", this means "managers" can perform all the operations such as SELECT, INSERT, UPDATE and DELETE.

The "customer" role is only granted the SELECT privilege on only the "transfer table" so that they can make new transfers or update existing ones, shown in Figure 8.

```sql
-- Roles
-- create the "manager" role
CREATE ROLE manager;

-- grant the "manager" role the necessary privileges
GRANT ALL ON account TO manager;
GRANT ALL ON loan_application TO manager;
GRANT ALL ON loan_payment TO manager;
GRANT ALL ON transfer TO manager;

-- create the "customer" role
CREATE ROLE customer;

-- grant the "customer" role the necessary privileges
GRANT SELECT ON account TO customer;
GRANT SELECT ON loan_application TO customer;
GRANT SELECT ON loan_payment TO customer;
GRANT INSERT, UPDATE ON transfer TO customer;
```

Figure 8 - Implementation of Database Roles in PostgreSQL.

Important Functions

Register_a_customer()

Figure 9, shows the implementation of the “register_a_customer()” function It takes five inputs and returns an integer. It starts by declaring a variable “customer_id” as an integer, it then inserts the input values provided, and retrieves the id of the customer and inserts a row and assigns it to the “customer_id” variable.

```sql
-- Function to register a customer
CREATE OR REPLACE FUNCTION register_customer(
    first_name varchar(255),
    last_name varchar(255),
    email varchar(255),
    phone varchar(255),
    dob date
) RETURNS INTEGER AS $$
DECLARE
  customer_id INTEGER;
BEGIN
  INSERT INTO customer (first_name, last_name, email, phone, dob)
  VALUES (first_name, last_name, email, phone, dob)
  RETURNING id INTO customer_id;
  RETURN customer_id;
END;
$$ LANGUAGE plpgsql;
```

Figure 9 - register_a_customer() Function.

View_balance()

Figure 10, shows the implementation of the “view_balance()” function, which allows This function allows customers and managers to view their account balance. The function takes an “account” number as a parameter. It then declares a decimal variable named “v_balance”. It then SELECTS the balance from the “account” specified if it matches and returns the corresponding value.

```
-- Function to allow the user to view balance
CREATE OR REPLACE FUNCTION view_balance(p_account_number INTEGER)
RETURNS decimal(19,4) AS $$
DECLARE
    v_balance decimal(19,4);
BEGIN
    SELECT balance INTO v_balance
    FROM account
    WHERE account_number = p_account_number;

    RETURN v_balance;
END;
$$ LANGUAGE plpgsql;
```

Figure 10 - view_balance() Function.

Apply_for_loan()

Figure 11, shows the code implementation of the “apply_for_loan()” fucntion,which allows the customer to apply for loans and enables managers to approve or deny such requests. It takes two parameters, an integer “p_account_id” and a decimal “p_requested_amount” which represents the ID of the customer and the amount of money they are requesting.

The function then inserts these values into the loan_application table. The current date is generated automatically, and the approval status and approved amount are set to “FALSE” and “0” by default.

Finally, it returns the ID of the loan application which was generated and saved in the table.

```
-- Allow the user to apply for a loan
CREATE OR REPLACE FUNCTION apply_for_loan(
    p_account_id INTEGER,
    p_requested_amount decimal(19,4)
) RETURNS INTEGER AS $$
DECLARE
    v_loan_application_id INTEGER;
BEGIN
    INSERT INTO loan_application (account_id, date, requested_amount, is_approved,
    approved_amount)
    VALUES (p_account_id, NOW(), p_requested_amount, FALSE, 0)
    RETURNING id INTO v_loan_application_id;
    RETURN v_loan_application_id;
END;
$$ LANGUAGE plpgsql;
```

Figure 11 - apply_for_loan() function.

Approve_loan().

Figure 12, shows the code implementation of the "approve_loan()", which allows the user to make transfers from one account to another. It takes three inputs: "sender's" account ID, "receiver's" account ID and the amount to be transferred. The function checks if the sender and receiver accounts belong to the same customer, if they do not the system will raise a condition, however if both inputs pass the function's check the function will update the sender's and receiver's balance accordingly.

```
-- Allow the customer to transfer amount to a different account under the same name
CREATE OR REPLACE FUNCTION transfer_funds(sender_id INTEGER, receiver_id INTEGER,
transfer_amount DECIMAL)
RETURNS VOID AS $$
BEGIN
    -- Start of the transaction
    BEGIN;

    -- Check if sender and receiver accounts belong to the same customer
    IF (SELECT customer_id FROM account WHERE id = sender_id) != (SELECT customer_id FROM
    account WHERE id = receiver_id)
    THEN
        RAISE EXCEPTION 'Sender and Receiver accounts must belong to the same customer.';
    END IF;

    -- Check if sender has enough balance to transfer
    IF (SELECT balance FROM account WHERE id = sender_id) < transfer_amount
    THEN
        RAISE EXCEPTION 'Sender account has insufficient balance.';
    END IF;

    -- Update sender's balance
    UPDATE account SET balance = balance - transfer_amount WHERE id = sender_id;

    -- Update receiver's balance
    UPDATE account SET balance = balance + transfer_amount WHERE id = receiver_id;

    -- Insert transfer record
    INSERT INTO transfer(date, customer_id, transfer_type, sender_id, receiver_id, amount)
    VALUES (NOW(), (SELECT customer_id FROM account WHERE id = sender_id), 'Internal',
    sender_id, receiver_id, transfer_amount);

    -- Commit transaction
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

Figure 12 - approve_loan() function.

References

Cabinet Office (2021). *E-Government Interoperability Framework (e-GIF)*. [online] Nationalarchives.gov.uk. Available at: https://webarchive.nationalarchives.gov.uk/ukgwa/+/http://www.cabinetoffice.gov.uk/media/254290/GDS%20Catalogue%20Vol%202.pdf.

Dybka, P. (2016). *Crow's Foot Notation*. [online] Vertabelo Data Modeler. Available at: https://vertabelo.com/blog/crow-s-foot-notation/.

IBM (2023). *Primary Keys*. [online] www.ibm.com. Available at: https://www.ibm.com/docs/en/iodg/11.3?topic=reference-primary-keys.

ICO (2022). *Guide to the General Data Protection Regulation (GDPR)*. [online] ico.org.uk. Available at: https://ico.org.uk/for-organisations/guide-to-data-protection/guide-to-the-general-data-protection-regulation-gdpr/.

Microsoft (2022). *Database normalization description - Office*. [online] learn.microsoft.com. Available at: https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description.

NHS (2023). *Government Data Standards Catalogue*. [online] www.datadictionary.nhs.uk. Available at: https://www.datadictionary.nhs.uk/supporting_information/government_data_standards_catalogue.html [Accessed 6 Feb. 2023].

PostgreSQL (2022a). *3.2. Views*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/tutorial-views.html [Accessed 6 Feb. 2023].

PostgreSQL (2022b). *5.7. Privileges*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/ddl-priv.html#:~:text=PostgreSQL%20grants%20privileges%20on%20some.

postgreSQL (2022). *Chapter 9. Functions and Operators*. [online] PostgreSQL Documentation. Available at: https://www.postgresql.org/docs/current/functions.html [Accessed 6 Feb. 2023].

Rabelo, J. (2019). *What is a Foreign Key? - Definition from Techopedia*. [online] Techopedia.com. Available at: https://www.techopedia.com/definition/7272/foreign-key.