**Product Evaluation**

Requirements from Part 1

* Users need to be able to sign in to their account on the software

As soon as the program starts, customers are asked to sign in to their account.

* The user must be able to change and/or delete their customer data

The user can access their Account Data, and then use this to edit their personal information, and banking information. They can also delete the tracked data about what songs they listen to.

* Customers can browse songs based on certain input criteria

Customers can navigate through certain predetermined queries e.g. Sort songs based on artists, reverse alphabetically. Originally the vision was to have the frontend be a website, where users could search for criteria e.g. search for songs starting with ‘s’, search for albums released on a certain date, but due to time constraints, this had to be lowered to a terminal application where such interactive interfaces are impossible.

* Results of any queries made to the database must be displayed to the user

This was mostly followed, it was decided that all select queries made would be displayed to the user through pandas, but any response such as “Query operated successfully” would not be displayed, but a custom message displayed instead.

* An intuitive interface for users to use

I believe this was achieved, when shown to other users on a trial run, there was no explanation needed on how to use the interface after about 30 seconds of them figuring it out, due to its simplistic, intuitive nature.

* SQL code should never be shown to the user

The customer never sees any direct SQL code, neither do they or can they ever type and SQL code to be executed.

* The user should never see a non-custom error message

In alpha testing, and the small amount of beta testing done, no errors have been thrown to the user that is not custom made.

* The program should never crash on the user

After debugging, no more bugs have been found that cause the program to crash

* Fast response time on the frontend

I have not noticed any extended response times, maybe with much more data, or a slower computer they may get noticeable, but except for startup, most response times are unnoticeable.

* This account data must be stored, and accessible to the user

The customer/banking/invoice data is stored inside the database, along with tracking on what songs customers listen to. This can be accessed by the user the data is on, at any time, and they can edit this (except the invoices) and delete all the information they want.

* All of the songs must be stored in the database, along with their respective attributes

All songs are stored on the database, and they all have their predetermined attributes of artist/s, genre/s, album etc.

* The database must be in third normal form to reduce data anomalies

The database is in third normal form, as can be seen in the ER Diagram, and the relational notation.

* The frontend python code must be able to query the backend SQLite database

The frontend imports the inbuilt ‘sqlite3’ library, in order to connect to and query the backend database using SQLite syntax.

* Must store all subscription data, on customers

All of the invoices for customers being allowed to use the service, are stored and can be accessed by the related customer at any time, but not edited.

* Data in the database must be validated using appropriate constraints

The backend database itself has ‘check’s in it so that the backend will throw an error if data of the incorrect format is ever inputted. The frontend also has error checking for any user input, to ensure even before reaching the database, that the inputted data is of a valid type, and the correct format

* All inputs should be validated, so there are no SQL injections

All inputs from the user, are validated so that no one can input data that could cause any SQL injections. They are validated by first off, most of them not being in a position to do anything, as they are all checked in python and then instead of the inputted string being used, they are checked against possible answers and then a predetermined SQL string is used instead. There are only four inputs that could be liable to SQL injection in the first place, the ‘New Bank Card Holder Name’, ‘New Password’, ‘New Username’, and ‘New Email’. The ‘New password’ input is hashed, and so is not able to be injected through. The other three are checked to have no spaces in them, no “, and no ‘.

* User data in the database is updated every once in a while, by asking them if it is up to date.

It has been decided that every six months, customers will be sent an email asking if their data is up to date and asking them to update it if it is not. This could not be implemented as any messages sent directly through the python script would be flagged as spam, unless a new email was created just for this, which due to time constraints could not be implemented.

* User data in the database is encrypted

This idea had to be scrapped, as in order to encrypt data, when the frontend is just a python script, the decryption key would need to be in plaintext, or easily found within the python script. Encryption would only be useful, if this program was hosted on a server and connected through that, but due to budget such a service could not be created. So encryption would be useless

* User passwords are stored as hashes

This has been done, the customer’s passwords are stored as sha256 hashes

* The code is clean, and easy to read

While the code does not have many comments, this is due to the mostly simplistic nature of the code, and the good practice of variable names and set up, along with the immense amount of print statements that describe to the user what is happening behind the scenes.

**Appendix**

**References**

ChatGPT. (2024, August 11). Name for the business. OpenAI. Prompt: “Here is the python code to create my SQLite database: conn = sqlite3.connect("SoundShift.db") cursor = conn.cursor() cursor.execute(''' CREATE TABLE Album ( AlbumID INTEGER PRIMARY KEY UNIQUE NOT NULL UNIQUE, Name TEXT NOT NULL, ReleaseDate DATE, CONSTRAINT Name CHECK (LENGTH(Name) <= 50) )''') cursor.execute(''' CREATE TABLE Song ( SongID INTEGER PRIMARY KEY UNIQUE NOT NULL UNIQUE, Name TEXT NOT NULL, Length INT, AlbumID INTEGER, ReleaseDate DATE, FOREIGN KEY (AlbumID) REFERENCES Album(AlbumID), CONSTRAINT Length CHECK (Length > 0), CONSTRAINT Name CHECK (LENGTH(Name) <= 50) ) ''') cursor.execute(''' CREATE TABLE BankDetails ( BankDetailsID INTEGER PRIMARY KEY UNIQUE NOT NULL UNIQUE, CardNumber INTEGER NOT NULL CHECK (LENGTH(CardNumber) <= 19), CardHolderName TEXT NOT NULL CHECK (LENGTH(CardHolderName) <= 50), ExpirationDate DATE NOT NULL CHECK (ExpirationDate > DATE('now')), CVV INTEGER NOT NULL CHECK (LENGTH(CVV) <= 4) ) ''') cursor.execute(''' CREATE TABLE Customer ( CustomerID INTEGER PRIMARY KEY UNIQUE NOT NULL, Username TEXT NOT NULL CHECK (LENGTH(UserName) <= 20), BankDetailsID INTEGER, Email Text NOT NULL CHECK (Email LIKE '%@%'), Password TEXT NOT NULL CHECK (LENGTH(Password) = 64), CONSTRAINT BankDetailsID FOREIGN KEY (BankDetailsID) REFERENCES BankDetails(BankDetailsID) )''') cursor.execute(''' CREATE TABLE RecentlyPlayedSongs ( BridgeID INTEGER PRIMARY KEY UNIQUE NOT NULL, SongID INTEGER, CustomerID INTEGER, DateListenedTo DATE, FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID), FOREIGN KEY (SongID) REFERENCES Song(SongID) )''') cursor.execute(''' CREATE TABLE Artist ( ArtistID INTEGER PRIMARY KEY UNIQUE NOT NULL, StageName TEXT NOT NULL CHECK (LENGTH(StageName) <= 50), FirstName TEXT CHECK (LENGTH(FirstName) <= 50), LastName TEXT CHECK (LENGTH(LastName) <= 50) )''') cursor.execute(''' CREATE TABLE Genre ( GenreID INTEGER PRIMARY KEY UNIQUE NOT NULL, Name TEXT NOT NULL CHECK (LENGTH(Name) <= 50) )''') cursor.execute(''' CREATE TABLE SongArtistBridge ( BridgeID INTEGER PRIMARY KEY UNIQUE NOT NULL, ArtistID INTEGER, SongID INTEGER, CONSTRAINT SongID FOREIGN KEY (SongID) REFERENCES Song(SongID), CONSTRAINT ArtistID FOREIGN KEY (ArtistID) REFERENCES Artist(ArtistID) )''') cursor.execute(''' CREATE TABLE SubscriptionInvoice ( InvoiceID INTEGER PRIMARY KEY UNIQUE NOT NULL, BankDetailsID INTEGER, CustomerID INTEGER, SaleDate DATE NOT NULL, AmountCharged NUMBER NOT NULL CHECK (AmountCharged > 0 AND ROUND(AmountCharged, 2) = AmountCharged), SubscriptionLengthBought INTEGER NOT NULL CHECK (SubscriptionLengthBought > 0), CONSTRAINT CustomerID FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID), CONSTRAINT BankDetailsID FOREIGN KEY (BankDetailsID) REFERENCES BankDetails(BankDetailsID) )''') cursor.execute(''' CREATE TABLE SongGenreBridge ( BridgeID INTEGER PRIMARY KEY UNIQUE NOT NULL, GenreID INTEGER, SongID INTEGER, CONSTRAINT SongID FOREIGN KEY (SongID) REFERENCES Song(SongID), CONSTRAINT GenreID FOREIGN KEY (GenreID) REFERENCES Genre(GenreID) )''') conn.commit() cursor.close() conn.close()

Write SQL code to add a bunch of records to the database. Have at least 100 songs, 10 customers, 20 artists, each artist having multiple songs, each customer having their own banking details, each customer having at least 1-2 subscriptions bought, each having played at least 5 songs recently, and 8 albums, each having at least 10 unique songs, all the appropriate bridging table data should be created, and have at least 5 genres. And connect every single song with a genre, and at least 20 with a second genre”

Emn178. (n.d.). *Sha256*. Online Tools. <https://emn178.github.io/online-tools/sha256.html>

Ian. (2020, May 8). *How to generate a random number within a specified range in SQLite*. Database.Guide. <https://database.guide/how-to-generate-a-random-number-within-a-specified-range-in-sqlite/>