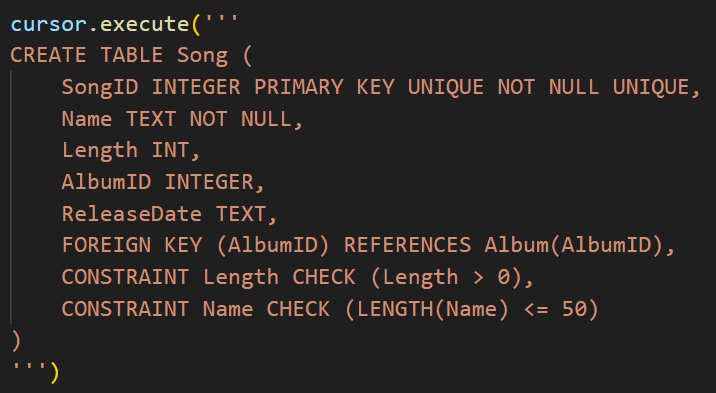
**Software Solution Features**

**Queries that:**

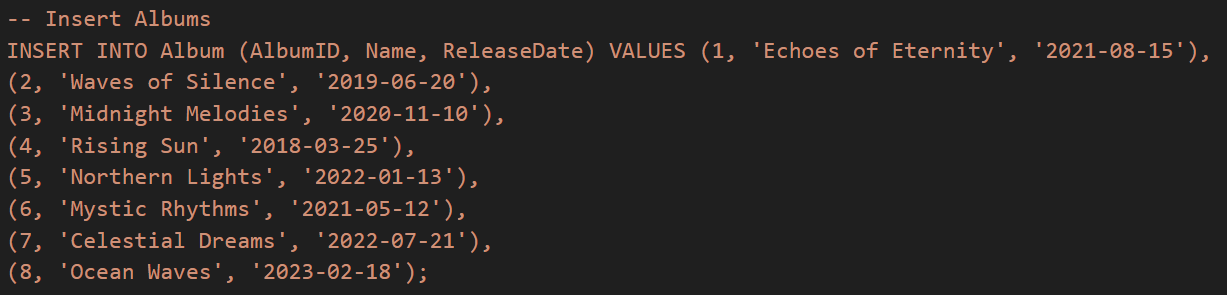
* Script that creates initial database

The “MakeSoundShiftDatabase.py” file first creates the database, or locates it if it is already there, and then deletes everything in it before having a multitude of queries that insert tables into the database. e.g.

this query creates a “Song” entity within the database, that enforces referential integrity, through its foreign key constraint on “AlbumID”

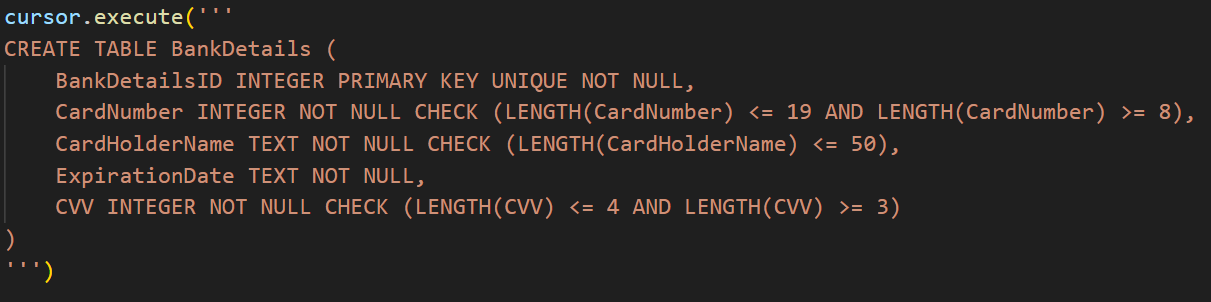
* Script that inserts data into database

The “PopulateSoundShiftDatabaseWithData.py” file connects to the prior created database, deletes all data within all the tables, before running 430 lines of SQL code that inserts an absurd amount of data into the database. e.g.



This code, inserts 8 records into the “Albums” entity, the extreme amount of data inserted into the entire database, allows for a large range of data to be tested

* Data validation within the SQL

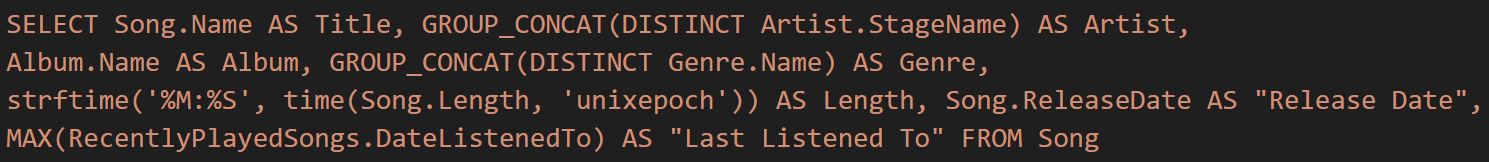
Each of the created SQL tables has constraints on them, usually in the form of checks. E.g. The Bank Details entity: 

The Primary key must be unque and not null; the “Card number” must not be null, and be a size between 8 and 19 digits; the “Card Holder Name” must not be null, and have a length less than or equal to 50; the “Expiration Date” must not be null; and the “CVV” must not be null, and have a size between 3 and 4 digits

This table can be inserted into, and its values changed, through the software solution, and even before being quiried into the entity. The code is too long to insert here but it lies within lines 545-633. This code checks: the “Card number” is an integer, and between 8-19 digits in length; that the “Card Holder Name” is less than or equal to 50 digits, is not null, does not include a “, or a ‘; checks the “Expiration Date” has a month between 1 and 12, and that the total date occurs after the current date so no expired cards can be inputted; that the CVV is an integer, and that it has a length between 3 and 4 digits

* Aggregate function queries

Nearly all of the queries have at least one aggregate function in them, the aggregate functions that have been used are MAX, SUM, COUNT, and GROUP\_CONCAT, this is four out of the six that exist. Examples where these have been used:



This uses GROUP\_CONCAT twice, and MAX

A screen shot of a computer

Description automatically generated

This uses MAX, COUNT twice, and SUM twice

A black background with white text

Description automatically generated

This uses MAX, SUM, GROUP\_CONCAT, and COUNT

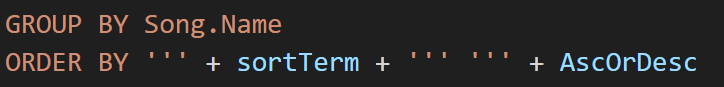
A black background with orange and white text

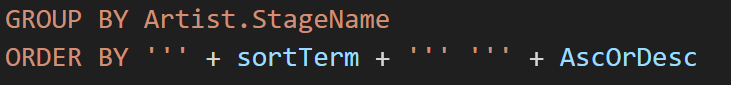
Description automatically generated

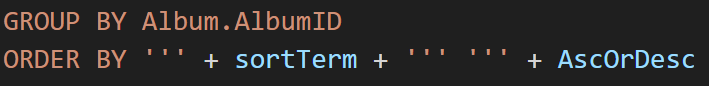
This uses COUNT twice, SUM, and MAX

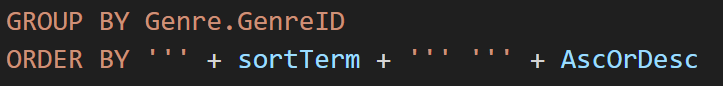
* Group By / Order By

These two particularly were used in all the customer browsing queries, with the ORDER BY quite often being preceded by variables, so that the user can chose what they want to order the table by: e.g.

 (Line 167)

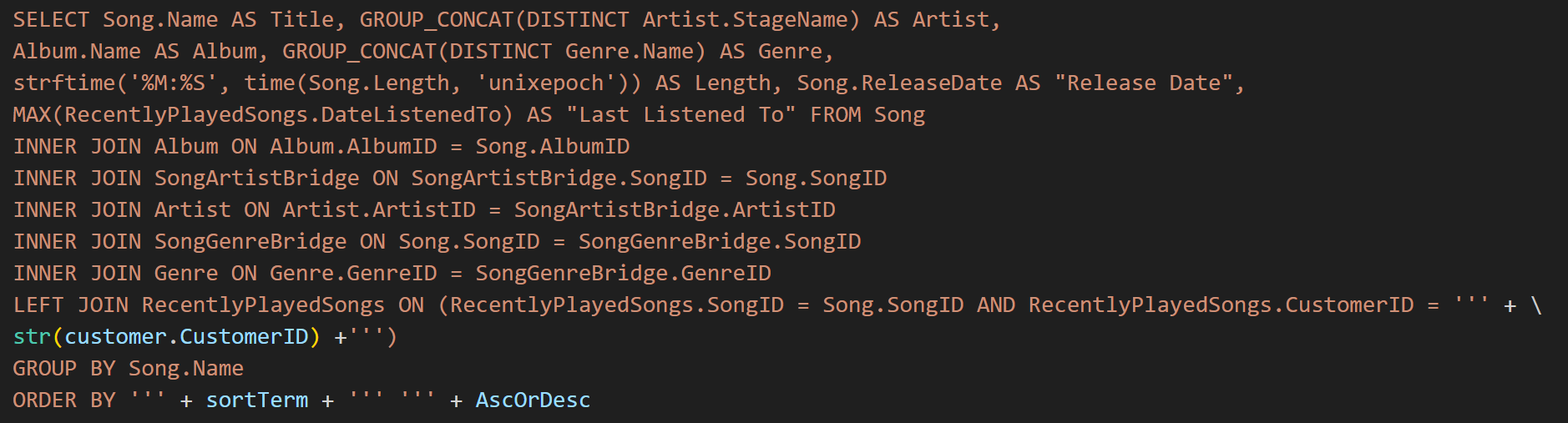
 (Line 225)

(Line 282)

(Line 338)

Each of these queries line up with the respective query up above e.g. the “Song.Name” query here is the end to the “SELECT Song.Name AS Title” query above. These “GROUP BY”s are necessary, due to the GROUP\_CONCATs present within the queries’ headers, which require GROUP BYs otherwise, all the data would be concatenated into one row

* Joins

This was used in nearly all if not all of the queries, and so only one example will be given, as they are all quite long: 

(Line 156). This has 5 inner joins, and one left join in it. The inner joins all only have the condition that the foreign key matches up, while the left join has the added condition, that it only joins when the specified customerID in the brdiging table is the same as currently loggedin customer’s customerID

* Calculated, and Concatenated fields with Aliases

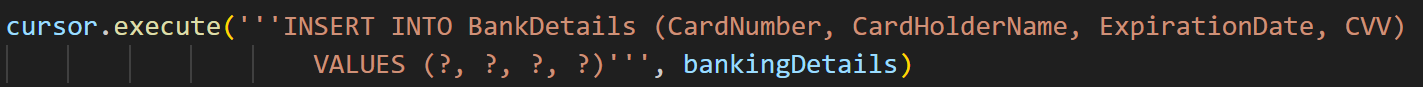
If you scroll back up to the aggregate functions queries section, you can see that all of the queries stated, have all of their fields given aliases. This amounts to 23 Aliases in those four queries alone.

In the second Aggregate query, the field: Is used which concatenates the artist’s “FirstName” and “LastName” fields together with a space in between to form the “Actual Name” field

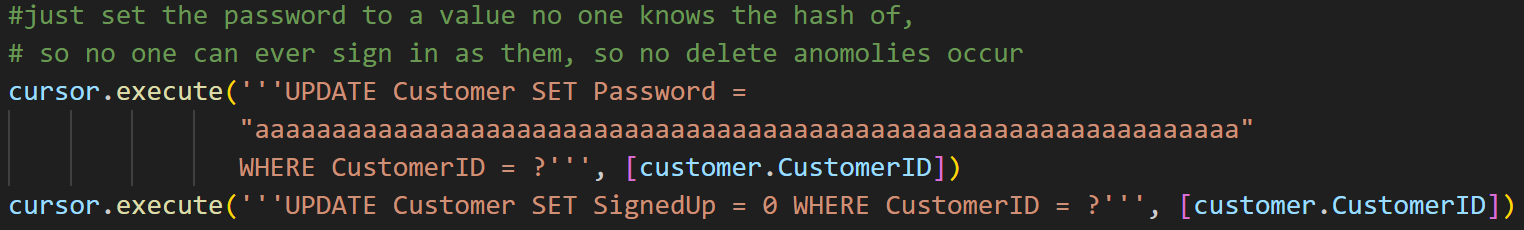
Some examples of calculated fields are:  Where the time of “Song.Length” (which is in seconds) is summed together, before this is changed into a time datatype, and then stringified in order to display the UNIX time stamp as an amount of minutes and seconds;

The field is also calculated, this is where the subscription length bought in an invoice (which is measured in months) is times by four, before rounded to two decimal places, as it is a monetary value, and a “$” is placed at the front.

* Insert, Update, Delete records

The Customer can insert a new banking details record. This then inserts a new record to the banking details entity. The user inputs all their data, and this is error checked before the query is run: 

The user can update records in a few instances, when they change bank accounts: 

When “Deleting” their account (the user is told they are deleting their account, but all that is really happening is the password is being changed to 64 ‘a’s. As this is the hash, the only way to log into someone’s account after this is to know the item that creates the sha256 hash of just ‘a’s. If this occurred, the hashing algorithm would be broken, as the hash was then reversible: 

When changing their username:



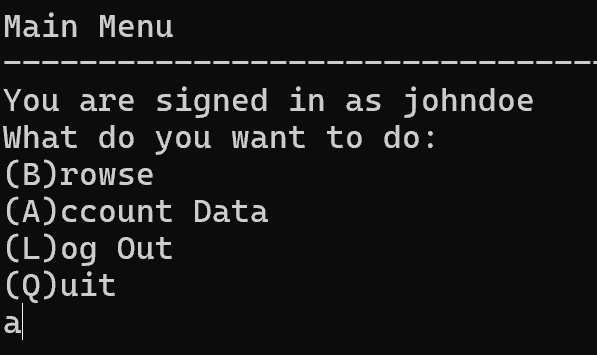
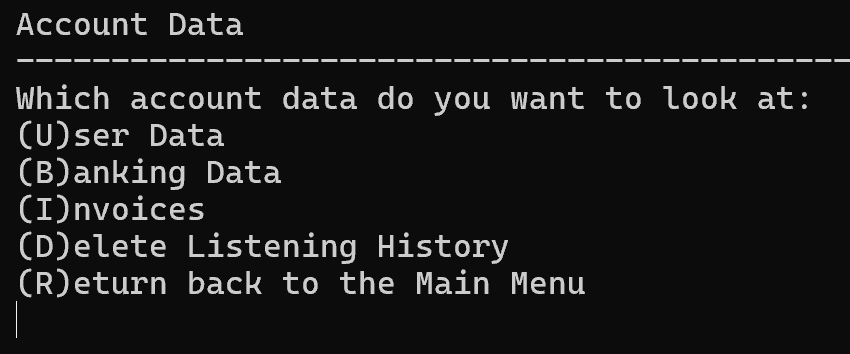
When changing their email: 

And many other instances.

The user can also delete the data tracking what they are listening to: 

**Extra Features**

* Interface for the user to update, insert and delete records

As spoken about above, the user can control the updating, insertion and deletion of records pertaining to them. They can control these actions through the frontend interface that us a terminal application, they can run and interact with, through the terminal. After signing in (use credential “johndoe” and “password”), users can navigate to “Account Data”, which opens up this screen: 

Users can enter “D” to delete all the data pertaining to them within the “RecentlyPlayedSongs” entity. If they enter “b” for banking data, they can enter “n” for creating a new bank account to pay with. After this they slowly enter the data one by one, before the record gets inserted: A black background with white text

Description automatically generated A black background with white text

Description automatically generatedA close-up of white text

Description automatically generatedA black background with white text

Description automatically generated

To update a record, instead of going into “Banking Data”, if the user went into “User Data”, then they could follow a similar proccess to just update their record.

* Suitable method of presentation of query results for the user

When the user searches through, and decides upon which of the predesignated queries to use, the output to the query is displayed to them through the terminal using pandas to look nice. The displaying is always the same as it is handled using the “DisplayQueries” function at line 85-109. The output looks like this: 

**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**

*Built-in aggregate functions*. (2023, May 12). SQLite Documentation. <https://www.sqlite.org/lang_aggfunc.html>

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/>