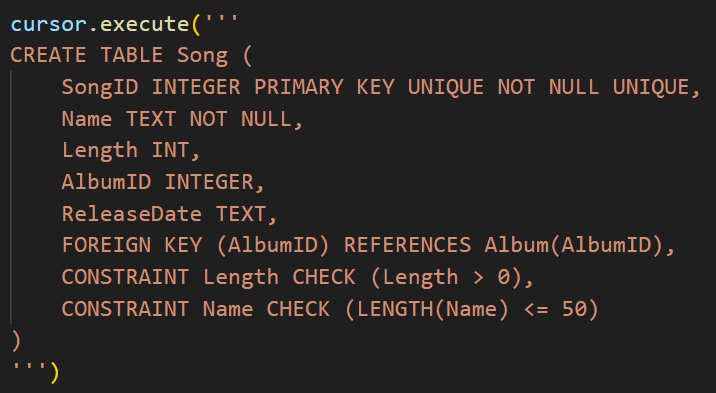
**Examples of Queries used:**

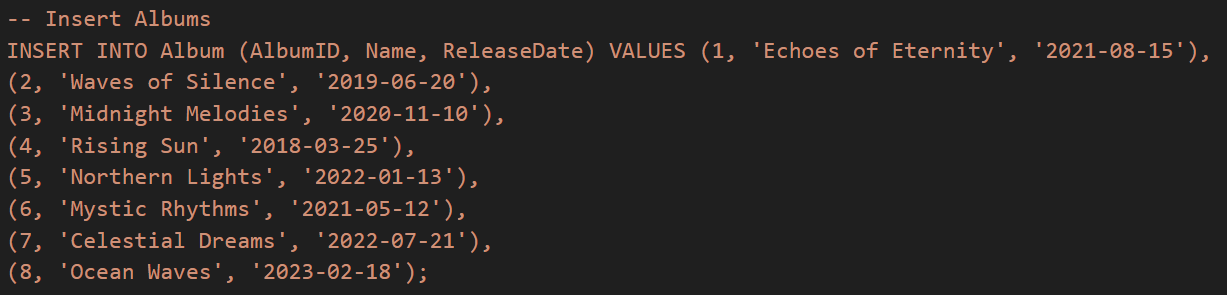
* Script that creates initial database

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”

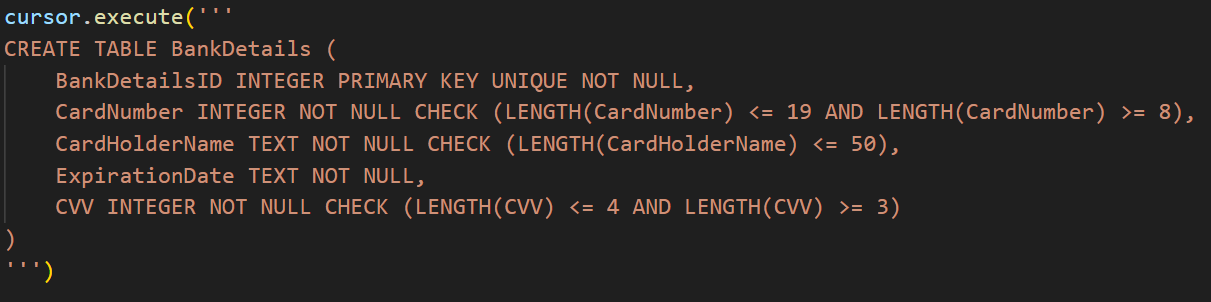
* Script that inserts data into database

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

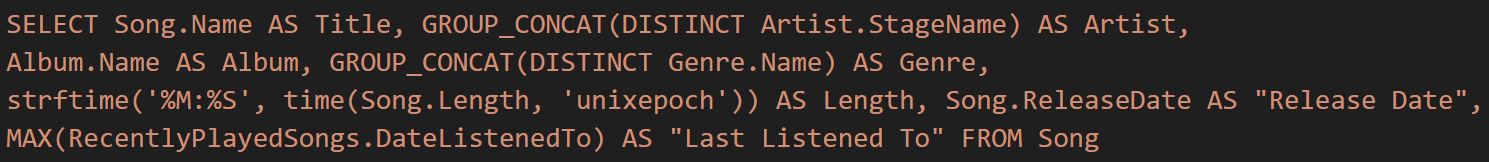
Each of the created SQL tables has constraints on them, usually in the form of checks. E.g. The Bank Details entity: 

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

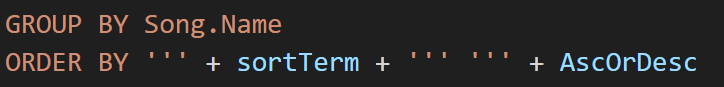
A black background with orange and white text

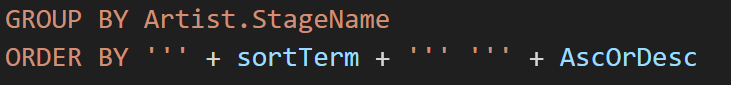
Description automatically generated

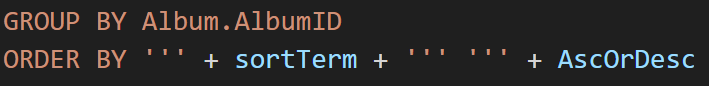
This uses COUNT twice, SUM, and MAX

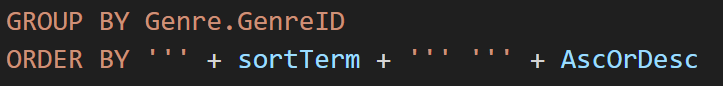
* Group By / Order By

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)

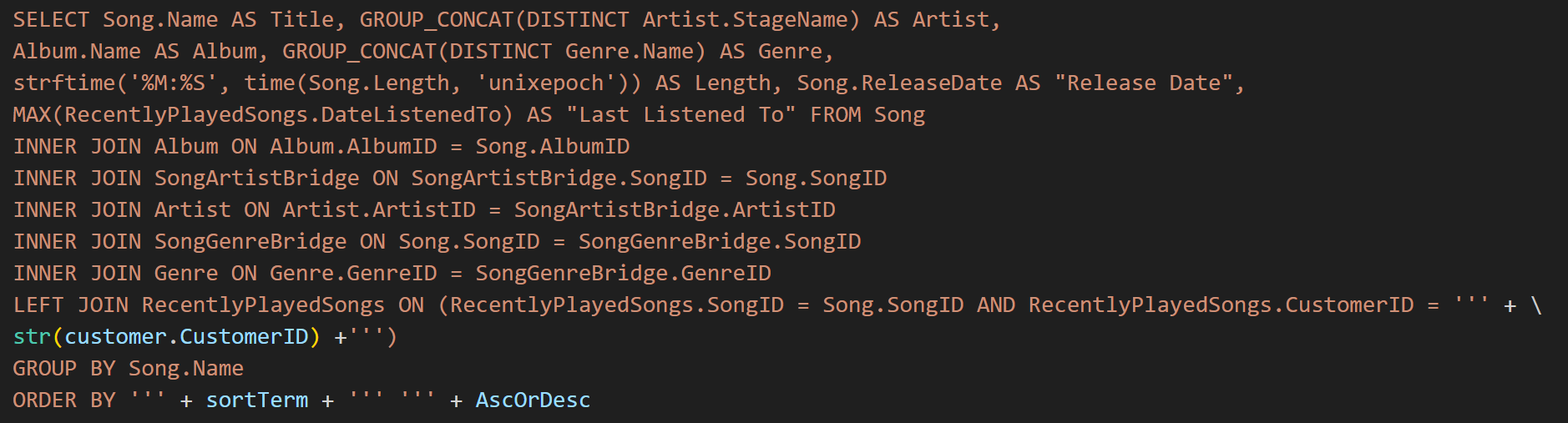
 (Line 225)

(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

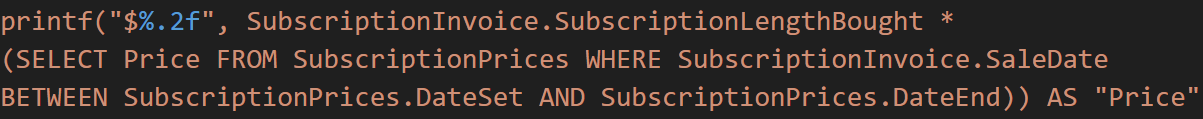
* Calculated, and Concatenated fields with Aliases

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.

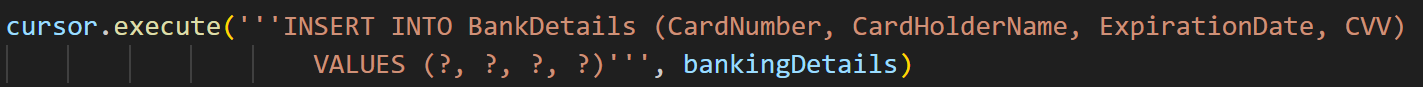
In the second Aggregate query, the field: Is used which concatenates the artist’s “FirstName” and “LastName” fields together with a space in between to form the “Actual Name” field

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 a length in minutes and seconds;

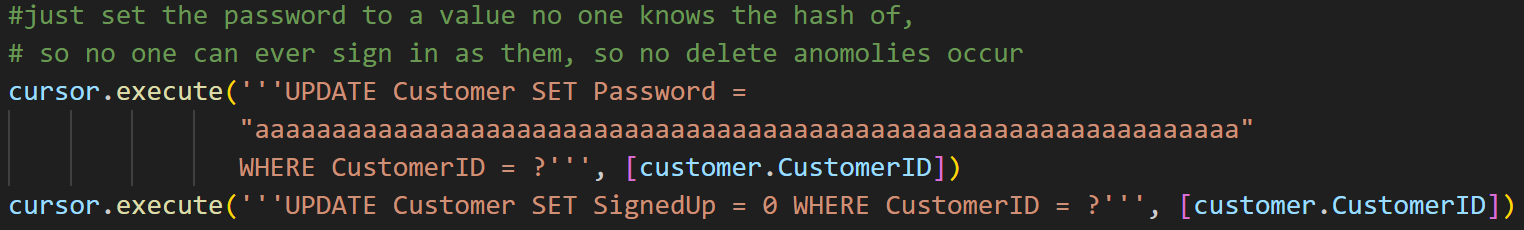
The Price field in invoices:

*(Line 655)* is also calculated, this is where the subscription length bought in an invoice (which is measured in months) is multiplied by whatever price the database saved it as being, during the time in which the invoice was purchased. Then this is rounded to two decimal places, and a “$” put in 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:



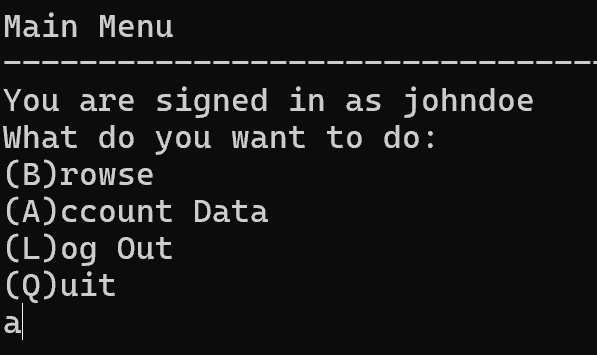
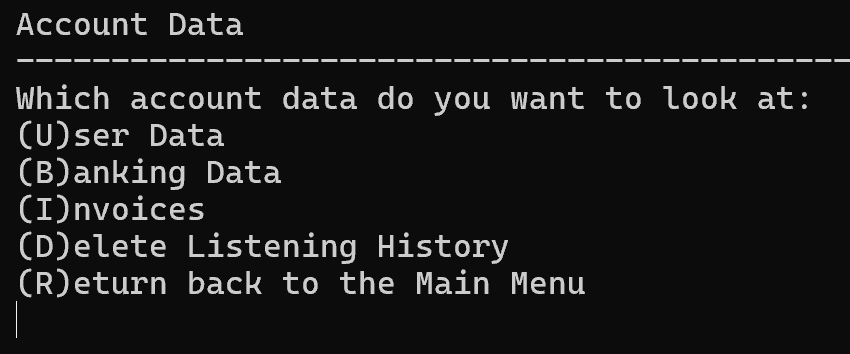
When changing their email: 

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

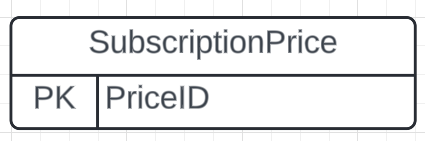
While the code does not have many comments, this is due to the mostly simplistic nature of the code, and the good practice of variable names and set up, along with the immense amount of print statements that describe to the user what is happening behind the scenes.

**Comparison of ERD diagram to finished Database**

Originally, I had the invoice just have a field storing how many months of subscription were bought by, and then just multiplying that by the current price for the subscription per month. This could lead to an update anomaly, if the price was ever changed, that would cause prior invoices to read the wrong price. To fix this, I added an entity called “Subscription Prices”. This entity stored the date that a price was set, and what price it was set to. Then whenever the Invoices were looked at, I would need to check which time range the invoice was bought in, and then multiply the monthly price at this time, by the number of months to get each invoices total price. Here is the database notation for this table:

SubscriptionPrices(PriceID, DateSet, DateEnd, Price)

As this new entity does not have any foreign keys, nor is connected directly to other entity, the ERD diagram stays the same, with a second ERD diagram just for this being there e.g.:



**Extra Features**

The extra features that were implemented were all within the frontend. These were features that were places within the Nonfunctional requirements outlined in part 1.

The output displayed to users using the front end, that was designed through pandas in python so that users could see the output returned from any sql queries that they asked for.

The logging in was an extra feature added, so that the users can edit their account data, and look at their recently watched songs.

The feature that passwords were hashed was not required, but due to the fact that it is industry standard to use shar256 to hash passwords, that was viewed as a required feature, even if it was no ‘required’ by the assessment.

The custom error messages that the user views after either an incorrect input, or if an actual error occurs was implemented, so that no user should ever have to look at a default python error message that means nothing to your standard person.

**Possible Improvements**

Some possible improvements to be made to the solution, is that a console interface is not professional, nor easy for your average customer to use. Ideally the front end would have been constructed using a python import such as tkinter or flask. This would have enabled, for tkinter, colourful images and buttons, an entire window making the solution into an app. This seems a lot more professional, as instead of a console application, it would be like every other app. Even flask, for an online application, where yet again buttons, and images, and graphic design could be utilised to make the application appear more appealing. This improvement could not be done due to time constraints, as with the time given, only a week could be allocated to the creation of the software solution, but in order to create fully fledged applications like these, much more time, possibly even months would have been required.

Another improvement would be to increase, the amount of functionality of the application. Currently the solution does not contain a way for new users to sign up to it. Nor does it contain a user-friendly method for publishers to publish their own songs upon the platform. And, the database is currently hosted locally, so every user has an individual version of it. Which makes the whole system take up lots of storage, instead of the standard model of hosting the database online, and then having the software merely be an API that connects to and streams from this online service.

**Known bugs and/or limitations and their impacts**

As, as soon as a bug was found, it was worked upon to patch up the bug. There are currently no known bugs present within the software. That is not to say there are no bugs, as every good developer knows, there are always bugs. But currently, no bugs have been found that have not been patched up.

Some limitations are that, currently the software solution does not actually check if someone’s subscription is still valid when they log in, along with the solution not automatically renewing someone’s subscription. Ideally, if a customer whose subscription had run out signed in, they would be greeted with a screen informing them that their subscription has ended and asking them to sign up to the service again. And, if a customer who had ticked auto renew subscription’s subscription was coming close to ending, then their subscription would be renewed, and their card automatically charged. Which leads on to the next limitation. Currently, the python application cannot first off check if any given bank details are correct beyond their format, but it is also not able to charge any customers on their given bank details. This was not implemented, as the solution is full of random credentials for sample data, and so as no proper credentials would be inputted yet, if the software checked if the data is correct, then all the sample data would need to link to actual bank accounts. The effect of the application not checking if a customer’s subscription has expired, is that even if one’s subscription is expired, then they can still sign in and use the service as normal.

**Retrospection**

The process followed to create this software solution, was that first, the Gantt Chart was created to set out a clear schedule of when everything should be done by, and a clear benchmark of how progress should go. This Gantt Chart was then dutifully followed, so that problem was outlined, requirements for the solution set out, and then legalities and ethicalities paid attention to. After that planning, the design for the database was drawn up with an ERD, before the database notation was set out. After finalising the design, the database was created in record time, and the frontend created which took the longest time. After finishing the frontend, this final document was made with some final adjustments to the frontend happening concurrently.

Some things that worked well: was that the setting out of what the software needs to be able to do, and then listing the requirements out and organising them as necessary and optional was extremely useful. This setting out requirements and their organisation is often a tactic used by both professionals and hobbyists including me, due to its extreme usefulness in creating clearly defined goals and expectations of what is to be achieved. And then the skill of needing to know which requirements must be scraped in the long run is a very effective skill to acquire and have. I will definitely continue using the requirement system as in the assessment, even if it would probably be used in a better system for lists such as a Trello board.

The planning out the database was very useful, and allowed me to spend time first off, worrying about normalising that database and creating everything that was needed for it, without needing to worry about SQL syntax, and any errors popping up. This planning procedure was very helpful, and it contributed massively to how quickly the actual database was able to be set up afterwards.

The final retrospection on if and how, the final solution met each of the predefined requirements was also very useful for checking that the set goals had been properly achieved. But in the future, with using a Trello board for the requirements, the checking of what requirements the application satisfies will be done concurrently to the development of the application, so that they shape the development instead of being slapped on at the end.

Some things that didn’t work well: was the retrospection, of how the ERD had changed, and going over what extra features were implemented. This is understandably a necessary part of an assessment, as this allows the marker to know that the developer actually knows what they are talking about, but in a actual project this would not be needed, and so would not be done.

The checking that the SQL queries checked all the criteria in the develop section of the assessment. Yet again, this is necessary so that the marker can ascertain that the developer actually knows SQL, and that the student included a wide range of SQL queries, and prompts, instead of just a few specific ones. This is necessarily there, but in an actual project this wouldn’t happen, as one wouldn’t check e.g. that their SQL queries use joins. They would just care that the software meets all the requirements, and functions properly. So, if this was an actual project, this step would be skipped as it wouldn’t be cared for when the time could instead be spend polishing the application that users are actually going to care about.

**What I would do differently next time**

I would change the requirements to be done on a Trello board (or a system like that), and then concurrently add necessary requirements to the list, as well as continuously check that the requirements are being met. Instead of forcing myself to write all the requirements before the project, and check which has been completed afterwards.

If this were to be done again, I would remove most of the Evaluate section, as that just resulted in report writing which mostly amounted to nothing but keeping the section on writing down the limitations of the software, as that seemed to be very useful in realising what work has to be done to the application before it would be ready to publish publicly.

The validation portion would also be removed, as it has just ended up taking lots of time for something, that in world would just need the documents to be submitted to GitHub, or if it is indie/a hobby, not needed at all.

**Appendix**

**References**

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

ChatGPT. (2024, August 11). Write SQL queries that inserts example data into the entities. 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/>