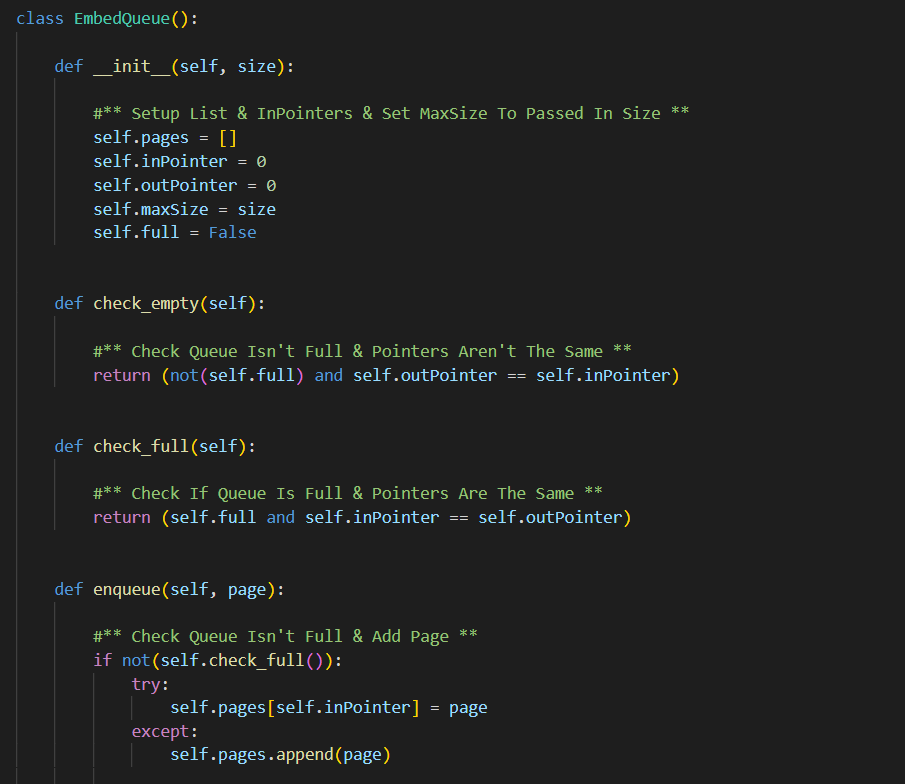
Logo

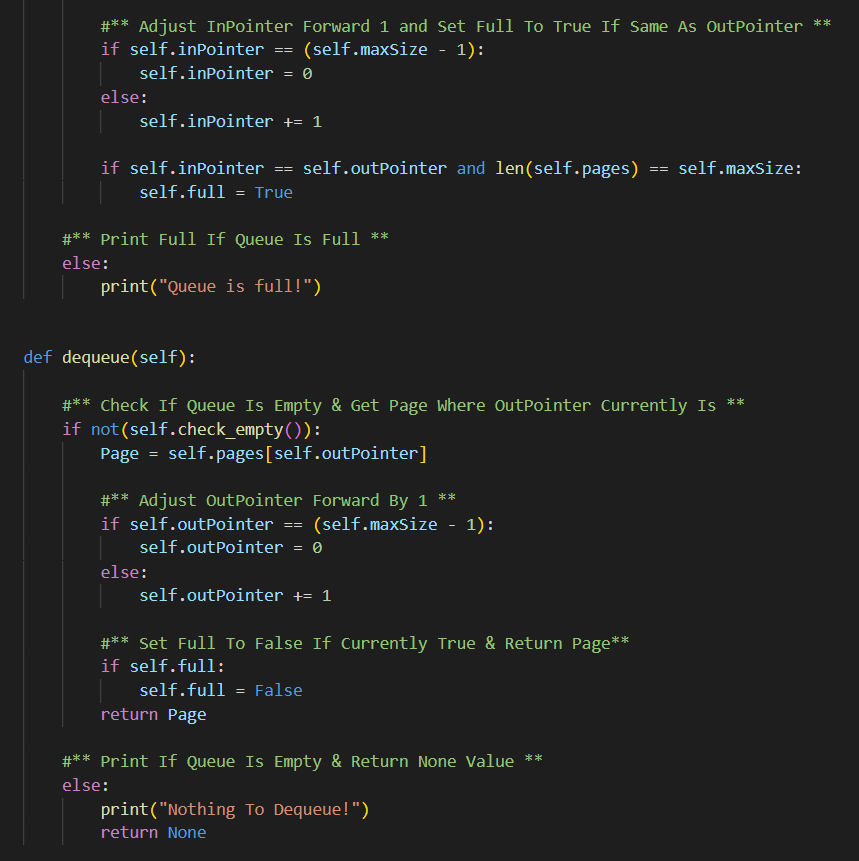
Description automatically generatedTechnical Solution

# Group A

## Data Structure – Queue

In my project, I have used 2 different queues, one which keeps track of a user’s listening history, and another that stores the different pages of embeds within the pagination module. A queue is a first in, first out data structure, where the first data put into the queue will be the first to leave the queue. Like how queues work with people, it forms an ordered, linear sequence of data, with two functions, enqueue, which adds new data to the back of the data set, and dequeue, which removes data from the front of the queue.





The queue shown above is used to store the different embeds, each representing a page, in embeds that need multiple pages, such as when returning song information or when showing user recommendations. It allows *maxsize* to be set at instantiation, where an empty list is set, with in and out pointers at 0, and an attribute called *self.full*, which is set to False as the list is empty. Each of the pages can then be added using the enqueue method, until the list is eventually full, when *self.full* will be set to True. Dequeue is used when pages need to be removed or replaced.

Text

Description automatically generatedText

Description automatically generatedThe queue is used by 3 functions in a separate class, which are called externally throughout other files in the code. The first is the *add\_pages(List)* function, which is called with a list of pages in order, and is where the queue is instantiated, and filled with all the pages in order. The first page is added to the back of the queue, as this has already been sent to the user, leaving the next page that needs to be sent always at the front of the queue.

The *get\_next()* function is a simple function, which moves the page at the front of the queue to the back and returns that page. This is called when the user requests the next page in an embed by reacting to the forward emoji underneath the embeds.

The *get\_last()* is a more complex function that dequeues and requeues all pages in the length of the queue – 1, with the last page to be requeued, being returned to the user, as this will be the last page that was previously sent to the user. This is necessary to retain the structure of the queue, as although I know the last page will be second from the back of the queue, the structure of the queue needs to be re-organised so that page at the back of the queue is brough back to the front as it is the next page.

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated

The second queue I used is found within the *Users.py* file and serves as the system used to store user’s listening history. I chose a queue to store the listening history as the first song listened to, will be the first song to be deleted from the history when the storage is full, so the queue’s first in, first out structure suits this perfectly. However, this queue needed to be altered slightly to keep track of the songs data, as well as just the song’s ID.

It works by using two arrays, *self.History*, which is a list of dictionaries returned from the database, storing data about the song, and when it was listened to, and *self.array*, which is an array of song ID’s. The pointers track the array of song ID’s, however, any changes made to that array are replicated on the *self.History* array, so that the data in both array’s matches at all times. The actual structure of the queue is the same as the other queue, with 4 functions, one to check the queue is empty, the other to check if it’s full, and both an enqueue and dequeue function for pushing and popping data. It also has a maxsize attribute, which is set to 20, and an attribute *self.full* to keep track of whether the queue is full or not.

Text

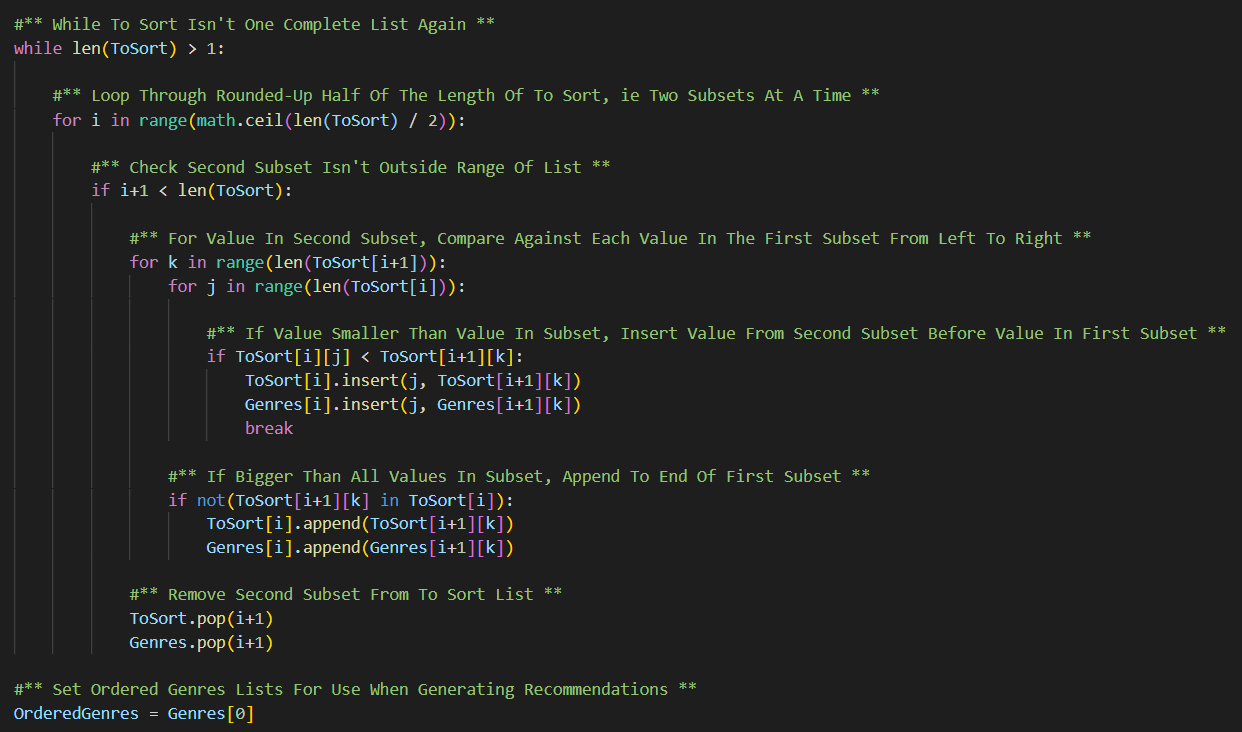
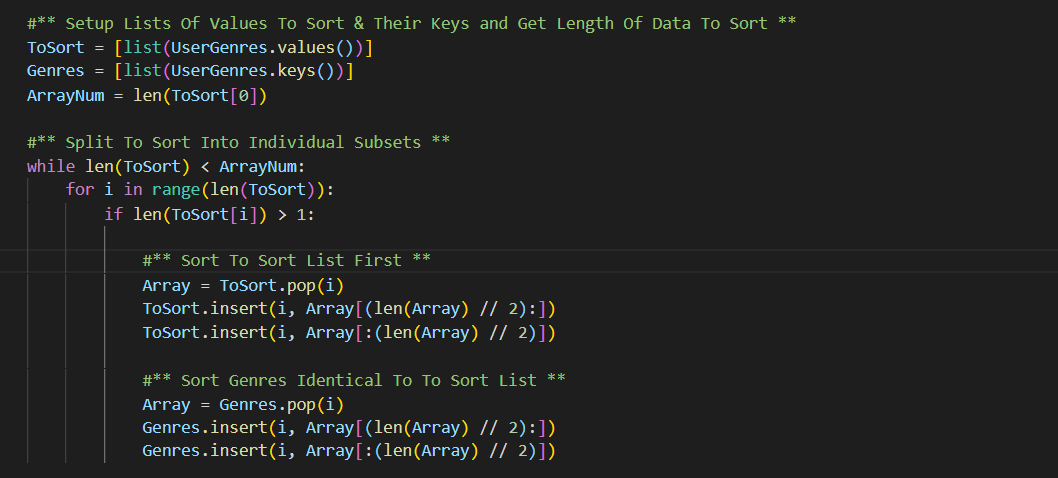
Description automatically generatedText

Description automatically generated

It's handled by a function in an external class called *incrementHistory(Song)*, that adds a new song to the queue. If a queue is full, a song is popped off the queue before the new one is pushed on. The function then gets the Spotify audio features for that song and adds those features into the average data used when generating recommendations, creating a new average, and adjusting the min and max values. Each time, the total songs listened to by the user is incremented by 1 as well, so it can be used for stats when showing their profile.

## Sorting Algorithms – Merge Sort

A merge sort is a sorting algorithm that involves dividing the data until you get all data individually separated subsets, before combining each subset together, 2 at a time, making sure the data inside each subset is ordered each time. This results in an ordered array of data, in a decently short amount of time. Its high efficiency is why I chose to include it in my project.

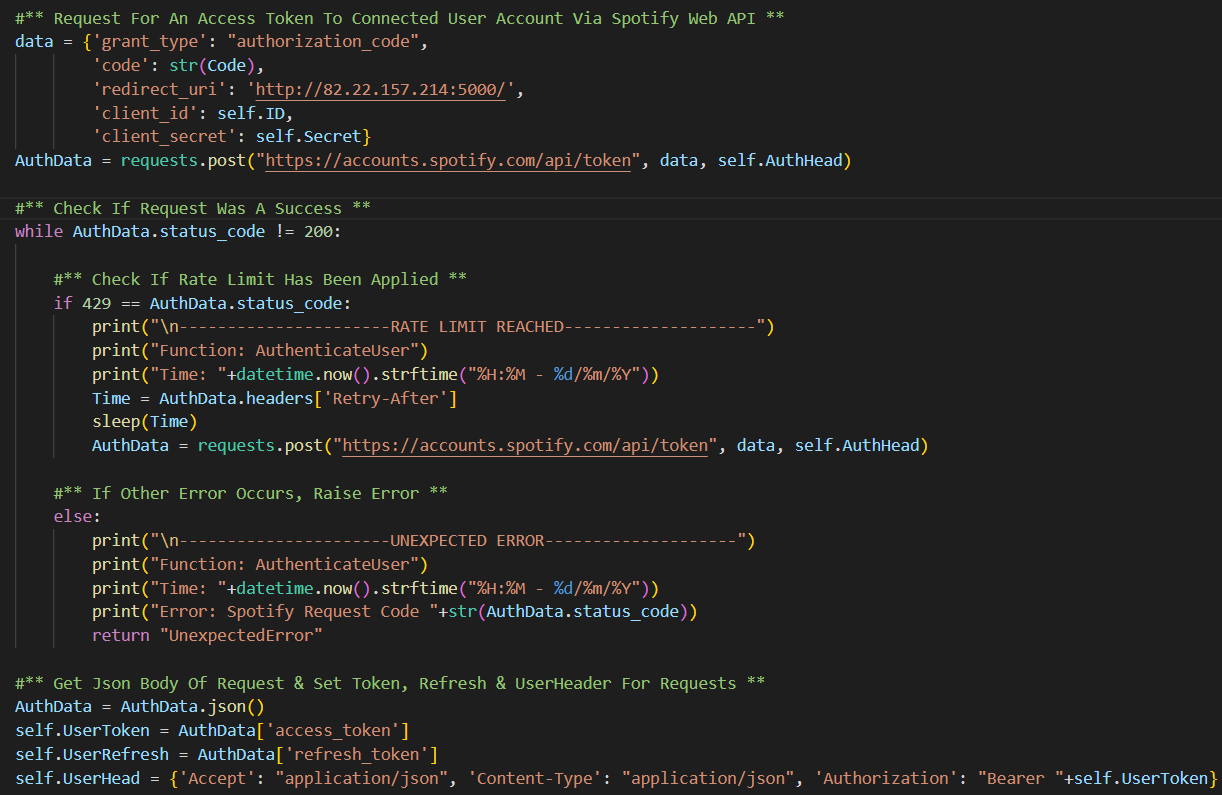


This merge sort is used within *RecommendFromTracks(List)* function in the Music class of the *MusicUtils.py* file. This merge sort works the same as a standard merge sort but uses 2 lists instead of 1. This is because dictionaries cannot be easily sorted by value. One list stores the keys of the user genres dictionary, and the other stores the values associated with each genre. As the values are numerical, this array is used as the primary array for sorting the two.

First, both lists are split into individual subsets, and then the values within the subsets from the values array are compared as the subsets are merged back together. Each time two subsets are merged and sorted; the movement of the values is the one array is repeated in the array of genre names so that each value is at the same index as that of the genre it is paired with in the other array. At the end of the merge sort, the values will be in descending order, meaning the genre at the front of the keys array is the genre that has the highest value in the original dictionary, and this is used to genre recommendations from the Spotify Web API.

## Using Parameterised Web Service API’s

APIs, often called RESTful API’s, are applications which allow different applications to communicate and share data. The term API stands for Application Programming Interface and are widely popular with many different companies using them to allow external applications to both collect data through a post http request and send data through a get http request, provided they are authenticated to do so. I used a variety of different API calls within my project to the Spotify Web API, for different tasks, such as authenticating users as well as collecting different bits of data about songs. These are parameterised based of different inputs into functions and return json data which is then parsed, and the useful data is abstracted and formatted before being returned.



Above is an example from the web.py file where I am calling the Spotify Web API, passing in a range of parameters. The request is found within the *Web.py* file and is used to send the received code post redirect from Spotify to get an access token for the authenticated user’s account.

It takes 5 parameters, the *grant\_type*, with is “*authorization\_code*”, as this is what is being used, the code in the URL when the user was redirected, the *redirect\_uri* that the user was redirected to, the Spotify client ID, and the Spotify client secret, which has been previously fetched from environment variables, and are stored as class attributes. These are passed in as a dictionary, sent using a post request to the Spotify Web API.

The status code of the request is then checked to make sure it is 200, representing a success, and if not, it is checked against other known common status codes. In this case, it is checked to see if it is 429, representing a rate limit being applied to the request. In this case, a time in seconds is passed into the received header of the request, of which must be waited for until the request can be repeated. The request is then retried.

If the status code is something not known, it is recorded as an unexpected error, and “*UnexpectedError*” is returned to where the function is called from. In this case, this would cause an error page to be displayed to the user, telling them they need to restart the authentication process. For all status codes, an appropriate error message is returned to console, letting the server admin see when and where the error has occurred in a request, and what status code is responsible, aiding a quick and easy fix of any potential issues with communications to the Spotify Web API.

A similar approach is adopted for all requests, with all other requesting being found in the *MusicUtils.py* file of the classes folder. Each request has a different set of parameters and tend to be get requests. Each request also has a different set of known, possible status code, such as 404 representing data not found, and 401, representing the bot’s token being expired, at which point, a function is called to generate a new token, before retrying the request. There are also a few requests to Spotify in the *Users.py* file, which use the user’s token, generated from the authentication process, to get stuff such as private playlists.

## Lists & List Operations

Lists are a data structure built into python designed to store data within an array of variable length. They are something I use often in my project, alongside dictionaries, and sometimes inside dictionaries. I then use list operations, such as iterating through the lists and using list functions such as *append(), pop(), len()*, as well as checking items are within lists, and getting their index within the list using the index function.

Text

Description automatically generated

The previous image is an example of list operations being used to get 3 random song IDs from a user’s listening history. Here a list is created from the *self.array* inside of the queue by going through each item and adding the SpotifyID to the list if it not none. Spotify IDs are then randomly popped from the list until the length of the list is equal to 3.

These tracks are then used as seeds when generating recommendations using the Spotify Web API. The list operations used in this example is *len()*, *append()*, *and pop()*, three very common and widely used list operations throughout the project. The *len()* function is a built in function which returns the length of a list. *Append()* and *pop()* are functions used to alter the elements within a list, with *append()* being used to add elements the end of a list, and *pop()* removing the element from the specified index. These are most used inside of the queues found in the *users.py* and *pagination.py* files.

Text

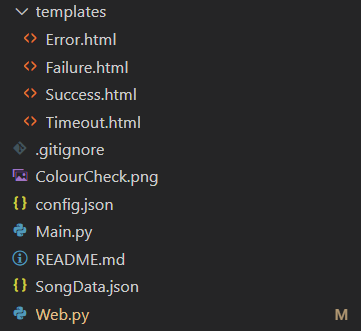
Description automatically generated

The second example is the use of a list iterator object being used within the History command of the *Account.py* cog, to loop through the songs in a user’s song history queue, concatenating the information about the song to the embeds description each time. First, I check the user has history by checking the length of the queue array from the *users.py* file. I then sort the list into ascending order using the In-Pointer from the queue, as the song in the position 1 less than this will be the newest song. I then increment i by -1, until it reaches the song in-front of the song I started with.

This works by using negative list indexing, where -1 represents the last item in a list and -2 represents the second to last, ect. I have used this to start adding songs from the back of the list once i becomes -1 and continue moving through the list from back to front. The result of this is a chronologically organised array of song history dictionaries. I then create an iterator object, that allows me to iterate through a list without the use of a for loop. This is useful, as in this case I am using the for loops to help create pages with 5 songs, before creating the next song, so having 1 big for loop through the list would make this difficult. I can therefore run the *next()* function on the iterator object at any time to get the next song in the original array without needing to keep track of the last index accessed.

The *next()* function will also return None if the iterator has reached the end of list, which I can break the above for loop, as I know there are no songs left. This is useful when there aren’t 5 songs to fill up an entire embed and keeps the algorithm efficient.

## Files Organised for Direct Access

Organising files for direct access is important as it makes both development easier, as well as improving readability and maintainability. In my project, I have separated my different python files into different sections based on what they are. In the main folder, I have both the main.py and web.py files, which must be manually run. This folder also contains the two json files, a temp image file, and all the other folders. My 4 files with classes that are imported as modules are found in the classes folder, and my cogs are found in the cogs folder. For the webpage, the 4 html templates are found in the templates folder.

I decided to organise my project code into different files and folders to help improve maintainability going forwards. By splitting my code across 12 different files, it means that the files can be organised into folders based on the functionality of the code. For example, all database calls are grouped within one file, *database.py*, with 1 call inside the *web.py* file, meaning that if there is an issue with the database, only 2 files need to be looked at rather than all 12 files, which would take much longer. This is similar with API calls to the Spotify API, with most API calls being found in the *MusicUtils.py* file, with a few in the SpotifyUsers class in the *Users.py* file.

Another benefit of organising my files for direct access is that they are easy to access from the code, as, for example, the web server must only need to look at the templates folder to find all the website templates it needs. This is the same for the classes, meaning no time is wasted by the program trying to look for the required file to load across the file structure. By splitting the commands across different cogs as well, it allows me to reload groups of commands rather than the entire bot, allowing me to fix errors without stopping users from making use of the commands within the bot. It also helps with organisation, as each cog represents a different group of commands, which is what is shown to the user when the run the help command, creating a more self-explanatory, easy to use bot and help command.

## Complex Data Model in Database

Text

Description automatically generatedFor my project, I have created a MySQL database with multiple interlinked tables to store information about users and cache data on songs. Databases are a dynamic data type that stores collections of data in organised, and structured tables. By using interlinked tables, different types of data about a user can be gathered at once from multiple different tables, speeding up the process of gathering data.

My database consists of 5 tables, with cache holding song data, and the rest storing different types of data about a user. Each table has a primary key, which is a unique identified for each record in a table. Some tables also have foreign keys, which are primary keys in other tables, allowing two records be easily linked cross-table. Each table is constructed using a series of columns to store different pieces of data, with each column having a different datatype. This datatype represents the type of data, and sometimes, the size of data, that can be stored in that column. It is important that the right type of data is always put into each column, otherwise, the insert statement will throw an error back, so I have had to carefully consider the type of data each column may encounter and therefore the best way to store it.

Table

Description automatically generatedTable

Description automatically generatedGraphical user interface, table

Description automatically generated with medium confidence

Above are 3 tables, the one on the left is the cache table, which has a primary key of SoundcloudID, and no foreign keys. It is designed to store both partially cached data, where only the SoundcloudID, SoundcloudURL, Name & Artists can be filled, which is used for songs where only Soundcloud data can be stored for a song, and full cache, which is used for Soundcloud tracks with accompanying Spotify data, where all columns are filled in a record. On each table, the datatypes are visible, showing what I have chosen to use for each column. The TINYINT represents Boolean data within MySQL.

The table in the top right is the start of the recommendations table. Here, I have chosen to show only the first 10 columns are the table itself has many different columns. The DiscordID is being used as a primary key in this table and is used to link to the primary key of the users table (bottom right) as these pieces of information are gathered using cross-table parameterised SQL (shown below), due to them being used together within the program. Discord IDs and Soundcloud IDs are completely numerical, however, are stored in varchar form, as some of these IDs exceed the maximum integer size allowed by MySQL. This means they have to be converted back to integers after they are retrieved from the database.

Graphical user interface, application

Description automatically generatedTable

Description automatically generated

On the left is the Spotify table. This table has a primary key SpotifyID, and a foreign key, DiscordID, which links the Spotify data with a user record in the users table. This table is used to store encrypted Spotify data from when a user decides to connect their Spotify account to their Discord account.

It is also used pre-linking to tell the web server that a person has asked to link their account via the !link command in the Discord first. This is done by putting in just a DiscordID & Linked timestamp. This timestamp is then used to make sure that the authentication hasn’t timed out. If a user tries to visit the website to link their account with a timestamp greater than 10 minutes ago, it will tell them they need to re-run the link command and delete the record.

On the right is the history table, which stores up to 20 records per user of listening history. It makes use of a compound primary key of both the DiscordID and ListenedAt timestamp. This is because it is impossible for the same user to listen to 2 different songs at the same time, so each entry will be unique. The SongID is a foreign key and links each record to cached data with the same SoundcloudID in the cache table. This means data about each song can be gathered from the cache table as each record is read from the database.

## Cross-Table Parameterised SQL

Cross-Table Parameterised SQL involves collecting different pieces of data from multiple interlinked tables at once, by using the INNER JOIN statement to collect data from rows in different tables which contain matching values. Within my project, I collect data from the users and recommendation tables at the same time based of the parameter DiscordID, and I also collect data from both the history and cache table at the same time based on having matching SoundCloud IDs between the two tables.

Text

Description automatically generated

The above code is the SQL query from the *GetHistory()* function in the *database.py* file. It uses an inner join between the SongID in the history table, and the SoundcloudID in the cache table to get song data at the same time as each record is fetched. It also uses ORDER BY command to get the returned records organised by most recent timestamp first, using the ListenedAt timestamp column in the history table. Only data that is necessary to be stored for songs inside the song history queue is fetched, rather than all the data in the cache record.

Text

Description automatically generated

The other inner join is used when getting user data from the users and recommendations tables together. It is found within the *database.py* file. Here all data from both tables is fetched from the database, as all data needs to be stored in the users class of the *users.py* file for the code to function properly. The data can then be formatted into a dictionary and returned to where the function was called from.

# Group B

## Dictionaries

Dictionaries are another type of data structure in python, used regularly in my project. They are arrays, like a list, but contain pairs of data, a key, and a value, and cannot be indexed. This means data can only be accessed by specifying the key, to fetch the value associated with that key. All keys must have a value associated with them and vice versa. Using dictionaries reduces the need to search for data, as it doesn’t matter the position of the data if you know the key associated with that data. This removes the need for searching algorithms. In my project, I use a lot of dictionaries to store data, particularly about songs. All json data also takes the form of a dictionary when within python, which I then must parse, and both my song data and config files are stored in json form.

Text

Description automatically generated

Text

Description automatically generated

The above is a function called *FormatSongData()* from the *MusicUtils.py* file. It is used by all Spotify Web API calls from the other functions within the class to format song data returned in json form from the requests, into a nicely formatted dictionary ready to be used by other functions within the code. It uses a mixture of lists and dictionaries to get the data into a suitable format. This involves placing all artists and their IDs into lists, as well as formatting the release date, album information, and adding “N/A” for songs with no popularity or explicit data. This is then formatted into the dictionary shown at the bottom of the function, with a key of the song’s ID, and the value being the data about that song. This allows the formatted song data dictionaries to be easily joined together for functions such as *GetPlaylistSongs(),* or *GetAlbumInfo(),* which have to get data on multiple different songs, but need to return all the data in one dictionary, hence the update function can be used to combine the dictionaries together before returning them, shown below.

Text

Description automatically generated

Dictionaries are also used elsewhere in the code when sharing data. In particular, the config file is json, and loads like a dictionary when used within python. User data is also stored in dictionary format within the *Users.py* file. When creating and working with track data in the *Music.py* cog, if the track has Spotify information associated with it, it is added as a dictionary of extra information within the track object, to be used by commands such as the nowplaying and queue commands.

## Records

Records are rows of data recorded within tables, for example in a database. I use these within the history row of my database, to represent the user’s listening history, which each record row linking a DiscordID to a SongID, which is linked to a record of song data within the cache table.

Graphical user interface, text, application

Description automatically generated

Above is a picture of 2 records within the history table of the database. Each user can have up to 20 records, with one record representing one song within their listening history. Each record links a user from the users table to a song ID from the cache table, with a timestamp of when the song was listened to. By separating the data into different records, it means each record only has 1 foreign key, and prevents any records being incomplete. It also means song history can be easily updated by only deleting the old records and adding the new ones, rather than having to re-format, and recreate one record with multiple foreign keys. This principle is used across all my tables within my database, with the split of data in records dependent on the primary key within the table.

Logo

Description automatically generatedTesting

Testing is an important part of creating software, as it allowed for the identification and rectification of errors before the project is made public. This is important as users will expect a reliable, and easy to use service, and constant errors can make a service difficult to use, losing potential users, which may result in both a loss of revenue and reputation for a given piece of software. It may also uncover any issues with the performance of the program, allowing you to try and speed up the code before any real user’s get hold of it, as they may be put off using it due to poor performance. The faster the program, the easier and more enjoyable it is to use.

Testing early on is increasingly important, as when software evolves, it tends to become more complex, making it more difficult to see any potential issues to the naked eye. It also allows you to check that you haven’t broken any older code whilst trying to add in your new features. Finally, it can check the security of your piece of software, making sure any potential security concerns are addressed before the software is made public. If there are lots of security issues within the software, it could lead to unauthorised access to your user’s data, which would be a massive hit to the reputation of your piece of software.

# Types Of Data Used

For my testing, I’ve used 3 types of data, normal, boundary and erroneous data. Normal data is data that is expected to be entered by the user, and when normal data is entered, the program should perform its intended functionality. An example of this would be entering a valid song link when using the play command, such as in test 2.0.2. Boundary data is data at either end of the allowed limits of possible inputs. An example of this is test 2.2.3, on the volume command, where the input of volume level must be between 0 and 100 as it relates to a percentage, so 100 is considered boundary data as it is right on the edge of the limit. The last type is erroneous data, which is invalid data. This could be anything from a mis-spelt input to a value outside the range of allowed values for a command. An example of this is test 2.0.9, where an invalid URL that has been mis-spelt, is entered by the user. This type of data will always result in an error, unless the program has some sort of autocorrect functionality to mitigate errors, for example, misspelling a song name when running the play command will just generate the closest result using the mis-spelt name rather than erroring.

# Testing Video

The testing video for the following testing table can be found on YouTube by visiting the following link:

**https://www.youtube.com/watch?v=KOe6lZorTwg**

*\*NOTE: The testing video does not follow the order of the testing table and jumps around between different groups of tests within the table.*

# Testing Table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Test Number | Test Description | Input Data | Expected Outcome | Actual Outcome | Steps taken to rectify issue | Time-stamp |
| 1.0 | Test bot sends message when joining new server. | None | Message is sent to highest text channel in server | Bot joins and sends message as expected in highest text channel | None | 0:18 |
| 1.1 | Ping command test | None | Message is returned showing bot’s latency | Returns message showing bot’s latency | None | 0:45 |
| 1.2 | Uptime command test | None | Shows a message telling user bots current uptime | Shows message correctly showing bot has been online for 1 minute | None | 0:51 |
| 1.3 | Invite command test | None | Displays embed with invite link for bot | Shows embed with invite link on it | None | 0:57 |
| 1.4.0 | Delete command with All | all | Deletes all user data from database | All user data cleared successfully from database | None | 42:56 |
| 1.4.1 | Delete command with Spotify | spotify | Deletes all Spotify data from database | All Spotify data deleted from database and account unlinked | None | 43:44 |
| 1.4.2 | Delete command with Users | user | Deletes all user data and listening history from database | User data isn’t successfully deleted from database, but bot doesn’t error | Slightly adjusted the SQL statement, and fixed some incorrect spelling of tables | 44:54  Fixed:  46:39 |
| 2.0.0 | Play command with no input. | None | Missing required argument error | Shows missing required argument error | None | 5:10 |
| 2.0.1 | Play command with plain text. | Take my breath the weeknd | Plays “Take My Breath” by the weeknd from SoundCloud | Joins VC & starts playing “Take my breath” by the weekend from SoundCloud | None | 2:26 |
| 2.0.2 | Play command with SoundCloud track URL | https://soundcloud.com/iamannemarie/anne-marie-beautiful | Plays “Beautiful” by Anne Marie from SoundCloud | Joins VC & starts playing “Beautiful” by Anne Marie from SoundCloud | None | 5:20 |
| 2.0.3 | Play command with Spotify track URL | https://open.spotify.com/track/2i6j4kdRKZV4Y62I2bD1QC?si=02d2cd44b4184824 | Plays “Back to the Future” by Bastille from Soundcloud | Joins VC & starts playing “Back to the Future” by Bastille from Soundcloud | None | 6:48 |
| 2.0.4 | Play command with SoundCloud playlist URL | https://soundcloud.com/lolomortar5000-official/sets/copyright-free-music | Loads all 17 tracks to queue, and starts playing first song, “Heroes Tonight” by Janji | Successfully loads all 17 tracks to queue, and plays through the queue correctly, starting with “Heroes Tonight” by Janji | None | 8:44 |
| 2.0.5 | Play command with Spotify playlist URL | https://open.spotify.com/playlist/4IK0dFDh925T60vsyfKJby?si=24572af2748a450a | Adds all 14 songs to queue, and starts playing first song, “This City” by Sam Fischer | Successfully loads all 14 tracks to queue, and plays through the queue correctly, starting with “This City” by Sam Fischer | None | 13:52 |
| 2.0.6 | Play command with Spotify Album URL | https://open.spotify.com/album/6CR4ozv4yOdaA3f6PPQepA?si=c8c47fed2cbe4422 | Loads into queue and starts playing tracks from specified album, 7 in total, from Soundcloud | Successfully loads all 7 tracks to queue, and plays through the queue correctly, starting with “Heroes Tonight” by Janji | None | 18:45 |
| 2.0.7 | Play command with HTML Stream Link | http://stream.live.vc.bbcmedia.co.uk/bbc\_radio\_one | Starts playing radio one from the web indefinitely | Joins, and starts streaming BBC Radio One to the VC | None | 23:26 |
| 2.0.8 | Play command with song link that relates to SoundCloud Go song. | https://open.spotify.com/track/79zGYOcAe2VmJsyQiJX31a | Let's user know only a preview is available and plays returned 30 second preview of requested song | Let’s user know it’s a preview and plays first 30 seconds before disconnecting | None | 26:57 |
| 2.0.9 | Play command with invalid Track URL | https://sondclod.com/iamannemarie/anne-marie-beautiful | Recognised invalid URL, and returns error to user letting them know there was a bad input | Gives error to use letting them know it was a bad URL as expected | None | 27:43 |
| 2.1.0 | Stop command when music playing | None | Music stops and bot disconnects from voice channel | Music stops as expected and bot disconnects | None | 33:30 |
| 2.1.1 | Stop command when no music playing | None | Let user know bot isn’t currently connected to a VC | Bot lets user know it’s not connected to a VC | None | 33:09 |
| 2.2.0 | Volume command with no input | None | Current volume level printed out | Volume level printed out correctly | None | 33:44 |
| 2.2.1 | Volume command with new volume within range | 60 | Volume set to 60% | Volume level set to 60% | None | 33:51 |
| 2.2.2 | Volume command with volume greater than 100 | 150 | Error telling user volume level must be between 1 and 100 | Error sent to user telling them volume level must be between 1 & 100 | None | 33:58 |
| 2.2.3 | Volume command with boundary data | 100 | Volume level set to 100% | Volume set successfully to 100% | None | 34:03 |
| 2.3.0 | Pause command when music not playing | None | Error telling user bot isn’t currently connected to a VC | Bot lets user know it’s not connected to a VC | None | 34:16 |
| 2.3.1 | Pause command both pauses and un-pauses | None (x2) | Music is paused when run first, then un-paused when run again | Music is paused, then un-paused again when the command is re-run | None | 34:28 |
| 2.4.0 | Skip command when music playing | None | Song is skipped and bot disconnects if end of the queue | Song skipped and bot disconnects as no other tracks in the queue | None | 35:04 |
| 2.4.1 | Skip command when music not playing | None | Error telling user bot isn’t currently connected to a VC | Bot lets user know it’s not connected to a VC | None | 34:47 |
| 2.5.0 | Queue command when songs in queue | None | Shows embed with list of songs in queue on it | Embed displayed with current playing track and songs in queue below | None | 35:29 |
| 2.5.1 | Queue command when queue empty | None | Prints message telling user queue is empty | Bot lets user know queue is currently empty | None | 35:11 |
| 2.6.0 | Shuffle command test | None | Queue is set to shuffled, proved using queue command footer | Queue is shuffled, shown using footer of queue command, and unshuffled when command re-run | None | 36:28 |
| 2.7.0 | Loop with audio currently playing | None | Current song playing is looped until command run again | Song is looped currently and restarts when skipped as expected | None | 37:20 |
| 2.7.1 | Loop with no audio currently playing | None | User is notified that it’s not currently connected to a VC | Bot lets user know it’s not connected to a VC | None | 37:02 |
| 2.8.0 | Seek with no input | None | Missing required argument error returned to user | Missing argument error correctly displayed | None | 38:34 |
| 2.8.1 | Seek with negative time greater than time passed in song | -60 | Song is seeked backwards to start of the song | Song seeked back to start of song and user notified | None | 38:02 |
| 2.8.2 | Seek with positive integer within remaining duration of song | 60 | Song is seeked forward the number of seconds provided by the user | Song is moved forward by 60 seconds successfully | None | 38:08 |
| 2.8.3 | Seek with positive input greater than remaining duration | 300 | Song is skipped onto next song in queue or bot disconnects if no more songs in queue | Song skipped, and bot disconnects as no more songs in queue | None | 38:20 |
| 2.8.4 | Seek with non-integer input | test | Bad input error is returned to user | Bad input error shown to user | None | 38:15 |
| 2.9.0 | Now playing when music is playing | None | Displays embed showing information about the currently playing song | Correct embed displayed to user, however, position field on embed remains at 0:00 despite position in song changing | Fixed hard coded value for position of track from 0:00 to take a value from *player.current.position* instead | 39:01  Fixed:  47:24 |
| 2.9.1 | Now playing when music is not playing | None | Let’s user know it’s not currently connected to a VC | Bot lets user know it’s not connected to a VC | None | 39:31 |
| 3.0.0 | Info with no input | None | Error returning letting user know they are missing an argument | Error letting user know they are missing a required argument | None | 40:05 |
| 3.0.1 | Info with Spotify Track URL | https://open.spotify.com/track/2i6j4kdRKZV4Y62I2bD1QC?si=02d2cd44b4184824 | Shows information about “Back to The Future” by Bastille in an embed with basic and advanced info pages | Shows correct information for requested song “Back to The Future” by Bastille in embed with both basic and advanced information | None | 39:43 |
| 3.0.2 | Info with invalid input | Song | Error letting user know they provided a bad input | Error letting user know input is not valid | None | 39:58 |
| 3.1.0 | Profile command without linked Spotify | None | Displays embed showing last song, last listening section and total songs listened to | Shows embed with listening history information for user | None | 40:17 |
| 3.1.1 | Profile command with linked Spotify | None | Displays embed including Spotify section with username and profile picture | Profile embed with section at top of embed with connected Spotify account information | None | 42:07 |
| 3.1.2 | Profile with no listening history | None | Displays embed with default values for last song sections | Shows default profile embed with default values for shown fields | None | 43:12 |
| 3.2.0 | History command with listening history | None | Shows user’s listening history in an embed with pages if more than 5 songs | Shows correct listening history across multiple different pages | None | 40:31 |
| 3.2.1 | History command with no listening history | None | Let's user know they need to start listening to some songs first | Tells user they need to start listening to songs first | None | 43:21 |
| 3.3.0 | Recommendations with no listening history | None | Let's user know they need to listen to some songs first | Tells user they need to start listening to songs first | None | 43:34 |
| 3.3.1 | Recommendations with listening history | None | Gives user a set of recommendations using their song history | List of 10 unique recommendations presented to user | None | 41:00 |
| 3.3.2 | Recommendations with Spotify | None | Gives user a set of recommendations using their Spotify playlists | List of 10 unique recommendations presented to user | None | 42:12 |
| 4.0 | Link command when unlinked | None | Embed is sent to user’s dm’s telling them how to link their Spotify account | Embed with how to link information correctly sent with valid link | None | 27:54 |
| 4.1.0 | User clicking on link sent by test command | None | User is redirected to Spotify for authentication | User instantly redirected to Spotify for authentication | None | 28:00 |
| 4.1.1 | User clicking on link after it’s timed out | None | User is displayed error page letting them know authentication has timed out | Error page correctly displayed when link visited after 10 minutes | None | 32:21 |
| 4.2.0 | User redirected after successful auth | User hits accept on Spotify | User is shown page letting them know it was successful, and data written to database | User correctly shown successful auth web page after being redirected from Spotify | None | 28:21 |
| 4.2.1 | User redirected after declined auth | User hits decline on Spotify | User is shown page telling them they declined it and their data is removed from database | User shown declined auth page and told to restart the linking process using !link in Discord | None | 31:09 |
| 4.3 | After auth confirmation in Discord | None | User receives successful auth embed in Discord DMs within 5 minutes | Successful auth embed received about 1.5 minutes after linking with correct information | None | 29:29 |
| 4.4 | Running link with a linked Spotify account | None | User receives embed asking them if they want to unlink their account & successfully unlinks | Received embed to unlink account & accepting successfully unlinks account | None | 29:36 |
| 5.0 | Help command main menu | None | Displays main menu embed showing possible command categories | Displays expected main menu with correct command categories | None | 1:07 |
| 5.1.0 | Help command with valid category | Music | Displays list of commands within Music category in embed | Displays correct embed list of commands for specified category | None | 1:22 |
| 5.1.1 | Help command with invalid category | Test | Let’s user know category / command doesn’t exist | Bot errors and fails to respond to message | Added error message when command or category isn’t found | 1:30  Fixed:  46:17 |
| 5.2.0 | Help command with valid command | Play | Displays embed showing info about the play command | Bot correctly displays information about play command | None | 1:45 |
| 5.2.1 | Help command with invalid command | Test | Let’s user know category / command doesn’t exist | Bot errors and fails to respond to message | Added error message when command or category isn’t found | 1:54  Fixed:  46:17 |
| 5.2.2 | Help command with admin-only command | Reload | Let’s user know category / command doesn’t exist | Bot displays embed with no information for reload command | Defined reload as a hidden command so the help command doesn’t recognise it | 2:05  Fixed:  46:27 |