Logo

Description automatically generatedDesign

# Tools I Will Use

I have chosen to code my project in python. This is because I plan to use data from Spotify to help predict the genre of a song, as this is something that Spotify’s Web API does not provide when requesting information about a Track. This is something particularly suited to the python language and is often used for similar functions such as machine learning.

## Modules

Throughout my project, I will be making use of many different modules to help me produce my discord bot. Some modules will be built in modules to python, some will be downloaded externally from Pypi, and a few will be created by me in the form of classes within separate files.

Built In Modules

* **Os:** *Provides functions to communicate with the operating system. I will be using it to get private keys stored in* *environment variables.*
* **Json:** *Module to help with the handling of json data. I will be using it when handling json body from requests, as well as when importing json data from* *my json files.*
* **Random:** *Used to help implement pseudo-randomization into my project.*
* **String:** *I will be using it to access a list of all possible characters for generating states using random when authenticating a user’s account through Spotify.*
* **Math:** *Provides me with a range of more advanced mathematical functions to use.*
* **Asyncio:** *One of the most important modules as it allows me to make my code asynchronous, meaning multiple functions can run at the same time. This is important as my code will need to be able to handle multiple different users trying to run different commands at the same time.*
* **Datetime:** *Used to get the current data/time, allowing me to keep track of the bot's uptime, amongst other uses.*
* **Requests:** *A module used to make requests to web pages and web API’s, such as the Spotify Web API.*
* **Base64:** *Functions for encoding and decoding characters to binary data and back.*
* **Copy:** *A module to help with both copying and deep-copying objects.*

External Modules

* **Discord.py (Discord):** *An asynchronous python framework used to make it easier to interface with the Discord API. Uses both functions and classes, along with decorators to help define commands, and has many other tools. It also handles all the raw requests to the API.*
* **Lavalink.py (Lavalink):** *Another asynchronous python framework designed to integrate Lavalink with Discord.py. Lavalink is a standalone audio sending node based on Lavaplayer and coded in Java. It is run separately as a lavalink.jar file and integrated into python through the module.*
* **MySQL.Connector:** *A python module provided alongside the MySQL server to allow easy connection to a MySQL database from python. Comes with easy to use execute and commit functions for SQL statements.*
* **Flask:** *A popular web framework for python. I will use it to create a small single paged web server where authenticated Spotify accounts are redirected once the authentication process is finished.*
* **Cryptography:** *A well known module used for encryption. I will be using it to encrypt user data fetched from Spotify when they link their accounts.*
* **Sklearn (scikit-learn):** *Well, known mathematical/science module. Used for creating a decision tree to be used when predicting genres of songs using data from Spotify.*
* **Pandas:** *Popular module for data manipulation and analysis. Allows for manipulating data into tables to easily analyze data to get min, max and mean values for data when generating recommendations.*
* **Dateutil:** *A module designed as an extension to the built-in datetime module, providing tools for me to easily calculate the time difference between two datetime values. Will be used for uptime and authentication.*
* **Colorthief:** *Small module which provides function that can be used to get the most prevalent colour in an image file.*

Custom Modules (Created by Me)

* **Database:** *A separate file which has a class called UserData inside which can be imported into other files. This class contains a range of functions that both update tables in the database, as well as return data from the database. Database connection is also established when this file is first imported. Can also return cursor and connection objects from MySQL.Connector should another function need to use these to execute a SQL statement not found within the class.*
* **MusicUtils:** *Another file with 2 classes inside. One class has all client calls to the Spotify Web API that* *do not require an* *authenticated user. Also has systems to refresh the ‘client credentials’ token when required as well as formatting data from the API into neat dictionaries ready to be returned. The second class is the main one that is called and inherits the Spotify class. This class has more complicated functions, such as the genre prediction and recommendation functions, and uses functions from the Spotify Class within these functions.*
* **Users:** *A file containing all user functions. It’s made up of 3 classes. The first is a SpotifyUser class which contains all the functions to get a user token for the Spotify Web API, as well as the user calls to the API. The SongHistory class is a queue data structure which holds a user's song history in the form of a queue of max length 20. The main class, Users, inherits both classes and has functions for both incrementing the song history, as well as saving the user object back to the database.*
* **Utils:** *A small file containing 1 class and some small functions. These are typically small functions that are used frequently in the code, such as formatting raw data. Any other miscellaneous functions are also within this module.*

# Database

For my project, I decided I needed a database to store information about my users, as well as storing data about songs I had fetched from the Spotify Web API. For users, I needed to store:

* *Basic Discord profile data*
* *Spotify Account Data & Tokens*
* *Listening History*

I would also need an additional table to cache information about songs, linking Spotify IDs with their equivalent Soundcloud ID, and storing the essential information fetched from *Get Track* API Call.

I chose to use MySQL as my database as it offered a fast and reliable, relational data structure, which would in turn help me to achieve my 8th user requirement. MySQL also offers a python module called MySQL Connector which offers a simple, well-integrated way of connecting my database to my scripts. It also offers an easy-to-use GUI called MySQL Workbench which gives me a straightforward way to monitor and adapt my databases as my project progresses. Another major selling point of MySQL is its security. As I am handling sensitive data such as Spotify access tokens, it is essential this is kept secure. MySQL makes use of the Secure Socket layer (SSL) as well as data masking and other layers of security to make sure data integrity is maintained, which is something I wanted to take advantage of.

## Database Tables:

***Users****(****DiscordID[varchar]****, Name[varchar], Discriminator[varchar], Avatar[varchar])*

***Spotify****(****SpotifyID[varchar]****, DiscordID[varchar], Name[varchar], Avatar[varchar], Followers[varchar],* *Subscription[Boolean], Refresh[varchar], State[varchar], Linked[date])*

**Recommendations***(****DiscordID[varchar]****, Min Popularity[int], Avg Popularity[int], Max Popularity[int], Min Acoustic[float], Avg Acoustic[float], Max Acoustic[float], Dance[float], Min Dance[float], Avg Dance[float], Max Dance[float], Min Energy[float], Avg Energy[float], Max Energy[float], Min Instrument[float], Avg Instrument[float], Max Instrument[float], Min Live[float], Avg Live[float], Max Live[float], Min Loud[float], Avg Loud[float], Max Loud[float], Min Speech[float], Avg Speech[float], Max Speech[float], Min Valence[float], Avg Valence[float], Max Valence[float])*

***History****(****DiscordID[varchar]****, SongID[varchar],* ***ListenedAt[date]****)*

***Cache****(****SoundcloudID[varchar]****, SoundcloudURL[varchar], SpotifyID[varchar], Name[varchar], Artists[varchar], ArtistID[varchar], Album[varchar], AlbumID[varchar], Art[varchar], Colour[varchar], ReleaseDate[varchar], Popularity[int],* *Explicit[Boolean], Preview[varchar], LastRefresh[date])*

Key -> Column[data type]

(Primary Keys are **Bold & Underlined)**

(Foreign Keys are Underlined)

## Entity-Relationship Diagram:

Diagram

Description automatically generated

# About The Code

This section defines my plan for the code for my project. When coding my project, I hope to explore object orientation, as well as achieving my user objectives as defined in my analysis. I aim to make the code reliable and easy to work with, setting up good error feedback for both the user and programmer, and keeping the code organized and well structured.

## Structure

The structure of code is very important. It sets the foundations for the project and helps make the code easier to both read and write. For my project, I wanted to use the object-oriented style of programming, where the code is organized around objects rather than just functions and logic. This has many benefits, including modularity of code, which allows for easier troubleshooting and better readability; re-usability of code, particularly useful if a section of code needs to store values; as well as the overall style of code making it much easier to design the code and put it together, allowing for overall better productivity. Within my code, I also plan to make use of inheritance, to help group together different classes, and reduce the amount of importing classes necessary when using separate files, whilst also expanding the number of functions available through a certain class.

Diagram

Description automatically generated

Above is a break down of the user created classes I plan to create for the project. Each one provides a good template, with all the necessary commands I’m going to need mapped out ready to program. Classes which have arrows going between them means that class is being inherited by the class the arrow is pointing to. The top section defines the class attributes set out during the instantiation of the class, with hard coded values written in where appropriate. The bottom section defines all command with the necessary parameters they’ll need to run. Each variable has a type associated with it in the form of varName:type.

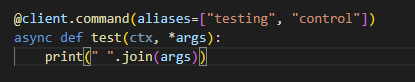
Functions are also a very important part of my project. Although my code will contain a few subroutines, a majority will be functions. My plan is to have almost all functions and subroutines contained within classes for easier readability, as well as giving me that modularity which will be particularly useful for the reliability of the code as only one object needs to be imported to get access to all functions, explained further down below. *Discord.py* also uses asynchronous functions to define commands. Commands are different functions available to the user, which all start with a predefined prefix, such as *!play.* When defining a command, a decorator is used.

Decorators are built into python and start with an At (*@)* symbol, sometimes called the “pie” syntax. It defines a function that should be called instead of the function, which will both call the function below the decorator, as well carry out additional things before, and after calling the function. Simply put, decorators wrap a function, modifying its behavior.

The decorator functions take the function it’s defined above as an argument automatically but can also have additional arguments defined. This is important as Discord.py uses decorators to define its commands. If a function should be a command, an asynchronous function is created, with an argument of ctx, and self if it’s within a class, and a decorator of *@client.command()* is added above or *@**command.command()* if inside a class.

Inside of the brackets of the decorator, different arguments can be passed in to alter different values about the command. In the project, I use the following arguments:

* **Aliases:** *A list of different names for the command which can also be used to invoke it by the user.*
* **Description:** *Simple and short description of what the function of the command is.*
* **Usage:** *Example of how to run command. Only used when the command has one or more arguments.*
* **Brief:** *Any additional information the user needs to know before running the command, for example, if the command needs a linked Spotify account to run.*
* **Help:** *To be displayed when a command has arguments and is a long description detailing all the command different possible inputs and what they each do.*



Above is a very simple command called test, which prints out any text written after the command into the console. It has 2 aliases, testing & control, which means it can be triggered by 3 different methods: *!test*; *!testing* & *!control*. The argument ctx is an object passed in automatically by the decorator when the command is called, and contains information about the command being triggered, such as the user who called the command, and where it was called from. This is a style of command I tested within my prototype, and I am planning to use many times within my final project. However, unlike my prototype, most of my commands will be sorted into classes, called cogs, and separated into different files, which Discord.py refers to as extensions. By splitting the command into cogs, the name of the cog becomes the name of that category/group of commands, and will help massively with organization of commands, as well as making it easier to work with, as a large file with 50+ commands would be long and very difficult to both troubleshoot and expand.

As mentioned, splitting my code into files will play an important part in the structure of my code, especially for the reliability of my code, which I talk about in the Reliability & Error Handling Section. Another major benefit is the productivity advantage of splitting the code into different files. This is because, for example, all music related functions are found within the *MusicUtils.py* module.

This way of splitting code into multiple categories means that the files don’t become long and difficult to work inside of, whilst helping with troubleshooting, as it’s more obvious where a particular function is located. My planned file structure is as follows:

* **Classes\**
  + **Database.py** *- Contains all functions related to the database and SQL statements.*
  + **MusicUtils.py** *- Contains all music related functions, including any functions communicating with music API’s.*
  + **Users.py** *- Any functions surrounding a user and user operations.*
  + **Utils.py** *- Generic functions that are used often but can't be categorized.*
* **Cogs\**
  + **Account.py** *- Commands about user accounts inside the bot, mostly using the users.py file.*
  + **Background.py** *- Any functions designed to run on a loop in the background whilst the code is running.*
  + **Help.py** *- All functions & commands designed to help the user with use of the bot.*
  + **Music.py** *- All commands & functions related to the playing of music through the bot, or information about music.*
  + **Pagination.py** *- A collection of functions designed to keep track of active embeds with multiple pages and change pages when necessary.*
  + **Utility.py** *- Any commands that are considered miscellaneous, but still aid the users experience with the bot.*
* **Main.py** *- The main file, containing any functions that must always be running, as well as establishing the connection to the Discord API.*
* **Web.py** *- A standalone file, which runs a small Flask web server, where users are redirected to after linking their Spotify accounts. Ran separately.*

Another important part about the structure of the code is how both global and local variables are used throughout the code. My main aim is to keep the use of global variables as small as possible as this will make my code run smoother. This is because global variables, like indicated in their name, are always stored in the program memory so they can be accessible at any point to any function. The main benefit of using classes is that multiple functions can share the same local variables within the class, which is something I will take advantage of throughout my code & classes. However, global variables will be used to store most constants within my *main.py* file, as although there is little code in the main file, it must be run from outside of a class meaning most variables are global variables. This is just part of *Discord.py* and the number of global variables in this file is kept to a minimum.

## Initial Startup / Code Execution

The startup of the code is centered around one file, named *main.py*. This file loads up all other files. In total, only 2 python files need to be run. To start the code up, first the lavalink server is loaded up by running the *start.bat* batch file within the Lavalink folder.

The *Main.py*, and *Web.py* are then each started up individually. At this point, the code should all be running, and any external connections should’ve been established. In the console of the *main.py* file, you can see each of the files slowly starting up, helping to indicate if any have failed to load. Here, you can also easily check that both the connection to the database, as well as the connections to the Discord API & Lavalink Server. The *main.py* file automatically loads all extensions and adds them to the bot. These extensions in turn import the classes from my “module” files, stored in the classes folder. A flowchart showing what should happen when the code is started up is shown below:

Diagram

Description automatically generated

As mentioned in the diagram, the config file is a small json file, which stores can be configured to easily change different variables of the bot when it starts up without going in and trying to edit the code. Inside the config file, you can configure the prefix of the bot; the welcome message, a set of text sent by the bot when it first joins a server; the status of the bot, and time between statuses, used by the background.py cog; the emojis the bot can use within its messages; and the active extensions, which defines the extensions which should be loaded by the bot on startup. The last one is the most important one, as it allows the bot to run minus a cog, should a certain cog need to be kept offline for extra maintenance, the bot can still load with that file unavailable, helping keep the bot’s uptime to a maximum. Some of the values within the config file, such as the statuses and status time can be easily updated by making changes to the file and reloading the background.py extension, whilst the others can only be changed on a complete restart of the code, as this file was mostly designed to be used once when the code is first run.

## Reliability & Error Handling

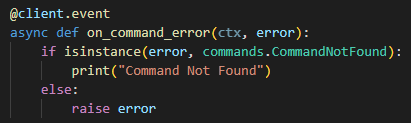
As the code is designed to perform as a server, reliability is super important. If the code stops, the entire service goes down, and users can’t use the bot anymore. We also want to keep uptime to a maximum possible, which means the number of times the bot needs to be taken down for Maintenance should be minimized. To help this, each of the files within the code can be reloaded separately via an owner only command located in the *Main.py* file.

The only file that cannot be reloaded without taking the bot offline is the *Main.py* file, as when this file stops, the connection to the Discord API is terminated, and the bot goes offline.

To make sure the *Main.py* file rarely has to be reloaded; minimal code is placed inside this file. The only code located here is either: code that is run once on startup only; the global error handling function; or any command that should be always accessible, such as *!reload.* The *Main.py* also deliberately doesn’t use any of the custom modules, so that these can be reloaded as well without having to reload the main file.

When reloading the python files, there are two different ways it must be done. For any of the cogs, Discord.py comes with a built-in reload feature for cogs, which will unload the file, removing all the commands inside the class from the bot, and then reload the file, adding all those commands back again. Inside each of the classes within the cogs is a function called *cog\_unload()* which is called before the file is unloaded, and I can use to help with cleanup. For modules, to make sure they are properly reloaded, I need to keep a record of which cogs use each of the modules and reload all those cogs instead. Because of this, reloading a module is more difficult than a single cog, and should be done only when necessary.

When handling errors, it is also important that these do not impact the running of the code, and that they are caught before they stop the python files from running. To do this, a global error handler located in the *Main.py* file is used. It’s located here as the only time the error handler is not loaded is when the entire code is not running. If the error handler was in a separate file, and that was reloaded, any errors that occurred during the reload time would cause the code to stop which we don’t want. Inside the error handler, I will catch all the generic errors that could occur during the code execution. Some of these are raised automatically by the *Discord.py* module, for example when a user runs a command with the wrong inputs; whilst others are self-raised during the execution of the code when an expected error occurs. However, only errors from the *Discord.py* module can be caught by the error handler. Other standard python errors would need to be caught by try and accept statements within the code. This is fine as most of the errors I will need to catch will be errors raised by *Discord.py*, hence the need for the global error catcher. The error handler also handles returning a message to the user in Discord to let them know what went wrong and how to prevent it happening again. This is important as I want to give the user as detailed feedback as possible to make sure they find their experience using the bot enjoyable.



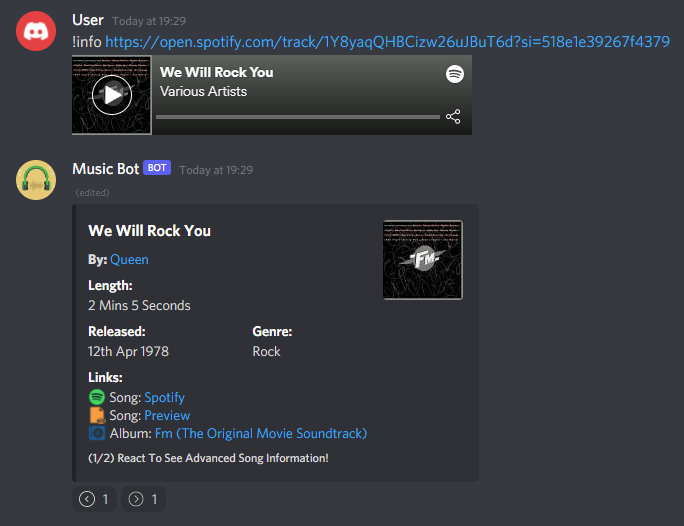
Above is an example of a basic error handler that I experimented with during my last prototype stage. This catches any errors raised during code execution and allows me to do different actions based on which error is raised. In this example, I am printing *“Command Not Found”* when *Discord.py* raises a *commands.CommandNotFound* error and raising any other error from the *Discord.py* would raise the error like what would normally happen if an error was raised in python and not caught by a try and accept statement. This is a concept I will expand upon in order to create a global error catching system, where all errors are caught and dealt with in one centralized function. This will allow for easier addition when a new error needs to be dealt with and will allow me to easily alter the error messages sent to the user and should help me to overall create a robust piece of code.

## User Interactions & Commands

Commands are a way for users on Discord to tell a bot to do something. These are defined by a prefix, followed by a string, which is the name of the command, for example, ‘*!test’*, would trigger a command called test to be carried out by any bot in the server that has the command prefix of *‘!’*. In the structure section, I talked about how commands were defined as asynchronous functions with decorators, and when a user runs a command, it triggers the function with the name or aliases which match the command name that has been called by the user. If the command name doesn’t exist, a *commands.CommandNotFound* error is automatically raised by *Discord.py*, which is handled by the error handler, as shown in the example in the Error Handling section. More than one command can be triggered by multiple users at the same time due to the asynchronous style of programming used within the project.

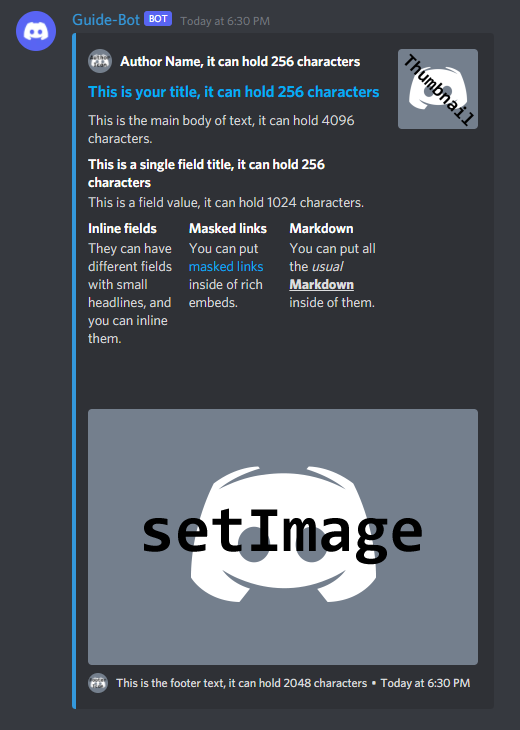
Commands can also have arguments, which must be passed in by the user to run the function, like how python works. These arguments are defined in the command function and must be added onto the end of the command call message. For example, a command defined as T*est(ctx, name, age)* would require the user to run *‘!Test name age’*. Each argument is classed as being separated by a space, and should one argument be longer than one word, it must be surrounded by question marks. Due to this, I don’t plan to do this within my project as it makes it very difficult for someone who doesn’t know how the bot works to figure out how to run a command. Instead, where a command needs to be defined with only 1 argument of variable length, I would use T*est(ctx, \*args)*, which means all text written after the command is separated into a list of words called args.

From here, I can join back together the arguments to create the original text that was passed in. By parsing the arguments on my side, I am removing any unnecessary complexity for the user which in turn improves an end user's enjoyment with the service I am providing.



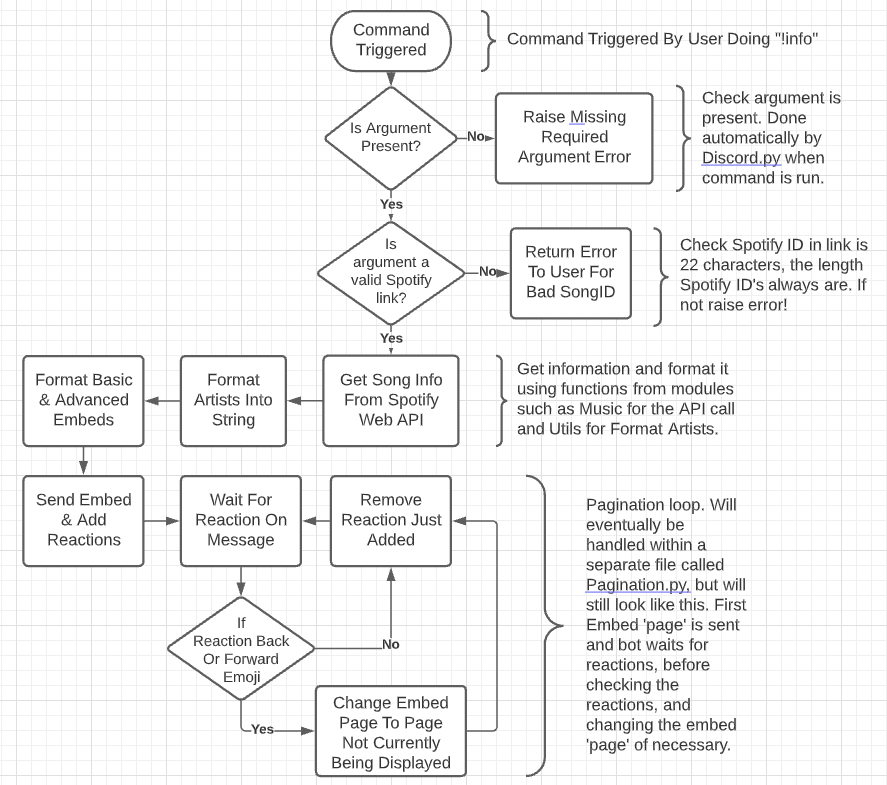
Above is an example of a user running the info command for the Queen song, *“We will rock you”*, providing the songs Spotify link as an argument. The bot responds with an embed, a type of image like object on Discord, which can be configured and sent only by bots. These embed are important as they provide a way for me to design how I present my data to make it as easy for the user to understand as possible. In the example above, the embed has been designed using sub-headings for each set of data on the embed, making it clear what each bit of data is referencing.

The embed above also uses pagination, a system allowing for multiple embeds to be grouped together like pages, and the reactions below the embed function as buttons that can be clicked to navigate through the pages. I plan to integrate pagination in a separate file called pagination, which will function as a global handler for pages, keeping track of all embeds with multiple pages and watching for reactions on any on them, adjusting the pages as necessary. Using one system for all reduces the likelihood of an error and makes it much easier to maintain and adjust later down the line.

Here is an example of an embed, and the different parts of an embed that can be configured, taken from [anidiots.guide](https://anidiots.guide/first-bot/using-embeds-in-messages/). In the other picture, the title is *”We Will Rock You”*; the description is *”By: Queen”*, where Queen links to the artist’s Spotify page; and the rest of the text are fields, besides the small text right at the bottom, which is the footer. I plan to use embeds a lot as they are a very neat way of presenting lots of information to the user, as well as allowing me to include pictures in a nicely formatted manner. However, I plan to use standard messages where only simple, small amounts of text are needed, for example, when returning an error to the user, or updating them on progress with a command. These are sent to the Discord API, through the *ctx.send(embed=Embed),* with Embed being the embed object defined within Discord.py, which is like the standard python print notation, however it must be awaited. The Discord API then constructs the embed seen above and displays it to the user. This method of communicating with a user is very similar to how you would communicate with a user through the console using print statements, which makes it easy to code and work with. 

## How Commands Work

What happens inside a command function is exactly what you would expect from a standard python function, however, all commands are coded as subroutines, using *ctx.send()* to send data to discord rather than returning values at the end of the function. This allows one command to send multiple messages, while using return would break the execution of the function after the first value is returned. Below is an example of what happens when the info command is called, one of the commands I created on my prototype:



This is the same command shown 2 images above this, and the flowchart follows what happens when the command is called by a user. Most of this happens within the function where the command is defined, and this is the same for most commands, due to the way the *Discord.py* framework works. One thing about embeds I plan to take advantage of is the idea of pagination. This refers to the use of multiple embeds to create ‘pages’, with different information shown on different pages. Each page is an embed, and users can change the page by reacting with emojis below the embeds. This isn’t built into Discord.py, so it must be set up and handled by custom code. This is very useful when you need to display lots of different things to the user, because embeds, like standard messages, have a character limit. This limit is shown on the embed example above the flowchart and is something that I must make sure I always stay underneath. Therefore, for commands like the queue command, which shows all songs in the queue, I need to use an adaptive amount to embeds to stay within the character limit, displaying for example 10 songs per embed. Pagination keeps the output clean, meaning that the servers text channel isn’t getting messy with multiple embeds. This is why pagination is very important and something I will be implementing on multiple commands.

## Creating The Web Server

As part of the project, I will need to create web server to handle authentication for Spotify accounts. This is because Spotify requires the user to be redirected to a webpage for the authentication code to be passed back to the server. To do this, the server needs to generate a random state, which must be stored on the server, and then the user needs to visit a generated URL, with parameters such as the scopes needed during authentication and the state that should be returned to the server with the code post authentication. I have decided I will need the following scopes for the bot to function as intended:

* **user-read-private:** *Allows bot to view further information about the account, such as the account’s subscription level. Doesn’t include email.*
* **playlist-read-private:** *Allows the bot to see the user’s private playlists. Necessary for allowing users to play their private playlists through the bot.*
* **playlist-read-collaborative:** *Includes collaborate playlists when getting a user’s playlists. Necessary for allowing users to play their collaborative playlists through the bot.*

It is important that no two active states are same, as this could mean that a user’s account gets linked to the wrong user. It is also important that the user starts the linking process in Discord first, so that each Spotify account can be linked to a Discord account. Both principles need to be considered when designing the system. For the structure of my web server, I will be using flask. Due to how flask runs; with an endless event loop, like how Discord.py works; it is easier to run the web server from a separate file. This means the web server will need to be started up and managed separately to the Discord bot.

## Keeping The Bot User Friendly

To keep the bot user friendly, I will create a help command which gives information about all commands and categories of commands within the bot. This help command will be organised in 3 ways, it needs a main menu, a list of commands for each category, and then a display of information for each singular command. The main menu needs to give information about the categories of commands, and how to use the help command. When specifying a category of commands, a list of details, including a brief description and the commands aliases, should be shown. For more detail about a specific command, the user should be able to specify a valid command to bring up a singular embed with as much detail as available about a bot.