Project report

Project title: Discord Music Bot

Course conductor: **dr Krzysztof Misztal**

Group members: **Krystian Jachna**

Introduction

Music has become an integral part of online communities, especially on platforms like Discord, where people connect to socialize, collaborate, or play games. The ability to play music in voice channels adds an enjoyable layer to these interactions, enhancing the overall experience. Personally, I often find myself listening to music during gaming sessions with friends on Discord, which sparked the idea for this project.

In recent years, many popular Discord music bots have become paid services or have been discontinued due to external restrictions, such as Google's enforcement of policies against bots that bypass ads on platforms like YouTube. This trend has limited access to free and reliable music bots, creating a gap for users seeking a customizable and sustainable solution.

Main Project Goal

The primary goal of this project is to develop a Discord music bot that is fully functional, and tailored to the needs of its users. The bot will include essential features such as:

a. Adding songs to a queue via commands.
b. Preparing songs in the background.
c. Playing songs sequentially after downloading.
d. Enabling continuous playback with a loop mode.
e. Controlling the music flow.
f. Sending embed messages in response to commands.
g. Multi-server support.

In addition, the project aims to incorporate tools like Docker to simplify deployment, enabling seamless hosting on external servers while also allowing users to easily host the bot locally.

Tools and Technologies

The following are the most important tools and libraries used in the project, representing only a subset of the technologies employed:

a. **Python 3.10**: Chosen for its simplicity and rich ecosystem of libraries, supporting asynchronous programming and API integration.

b. **discord.py**: A library for interacting with the Discord API, enabling commands, voice channel management, and user interactions.
c. **yt_dlp**: A fork of youtube-dl used for downloading/streaming music from platforms like YouTube.
d. **youtube_search**: Added to perform keyword-based searches for music on YouTube.
e. **asyncio**: Used to handle concurrent tasks such as downloading music, managing queues, and processing commands.
f. **Docker**: Provided a consistent runtime environment by containerizing the bot and its dependencies.
g. **Google Cloud**: Hosted the Docker image for scalable and reliable deployment.

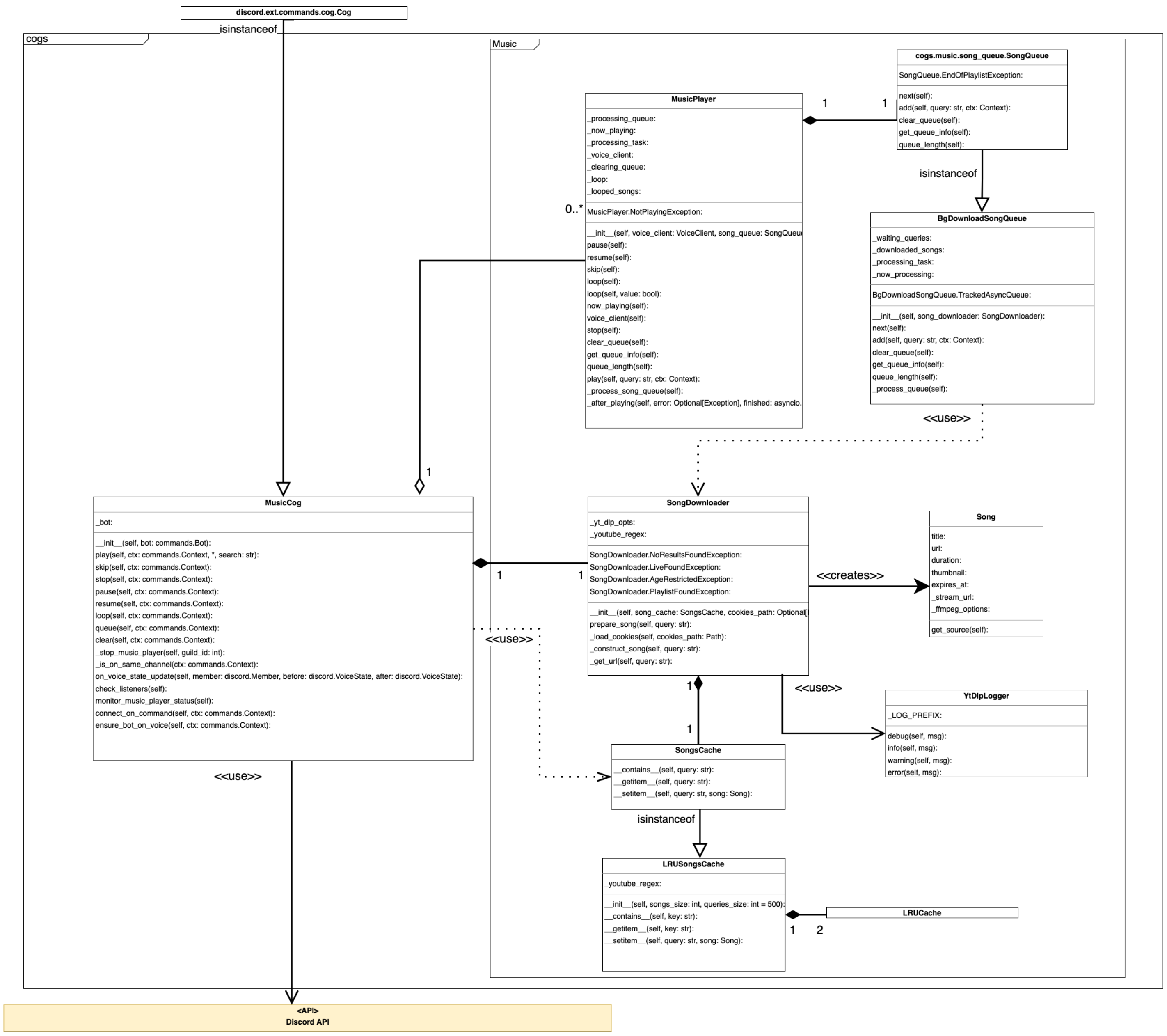
System Desing
The Discord Music Bot is built with a highly modular architecture, utilizing Cogs to separate concerns and enhance flexibility. At the heart of the system lies the *MusicCog*, which extends the *commands.Cog* class from Discord.py, and serves as the main interface for handling user commands related to music playback. The *MusicCog* incorporates various functionalities for playing, pausing, and stopping music, and it manages commands for controlling the playback queue in a user-friendly manner.

The *MusicPlayer* class is designed to handle the core logic for managing the song queue and the actual playback of music tracks. This includes responding to commands such as play, pause, skip, and stop. The player processes music asynchronously to ensure that it doesn't block the bot's ability to respond to other commands. The song queue is implemented using the *SongQueue* class, which stores the songs that will be played next. To enhance the bot's performance and ensure seamless playback, the bot includes the *BgDownloadSongQueue* class, which serves as an advanced form of the *SongQueue*. This queue preloads songs in the background to ensure that songs are ready to play as soon as the current song finishes. The *BgDownloadSongQueue* works by utilizing the *SongDownloader* class, which is responsible for fetching audio files from external services and preparing them for playback. This preloading process ensures that there are no delays when transitioning from one song to the next.

The *SongDownloader* class is integral to the bot's operation, interfacing with external APIs and services to download music files. It not only retrieves the audio files but also creates *Song* objects that encapsulate metadata such as the song’s title, artist, duration, and URL. To maximize efficiency, the *SongDownloader* incorporates a caching mechanism in the form of an *LRUSongsCache*. This cache stores recently downloaded songs, reducing the need for redundant downloads and improving overall performance. The caching mechanism ensures that once a song is downloaded, it is available for future playback without needing to fetch it again.

To manage the interaction with the Discord voice channels, the bot uses the Discord API, which enables it to join and leave voice channels dynamically. It also handles sending the audio stream to the appropriate voice channel and reacting to various user commands in real-time. The modularity of this design allows for easy updates and improvements to individual components of the system, such as switching out the *SongDownloader* or adjusting the behavior of the music queue. The bot is designed to be highly extensible, making it easy to add new features or modify existing ones as needed.

A detailed UML diagram illustrating the architecture of the bot is presented in Figure 1, which provides a visual overview of the system's components and their interactions.



discord.ext.commands.cog.Cog
isinstanceof
cogs
Music
cogs.music.song_queue.SongQueue
SongQueue.EndOfPlaylistException:
next(self):
add(self, query: str, ctx: Context):
clear_queue(self):
get_queue_info(self):
queue_length(self):
MusicPlayer
_processing_queue:
_now_playing:
_processing_task:
_voice_client:
_clearing_queue:
_loop:
_looped_songs:
MusicPlayer.NotPlayingException:
__init__(self, voice_client: VoiceClient, song_queue: SongQueue
pause(self):
resume(self):
skip(self):
loop(self):
loop(self, value: bool):
now_playing(self):
voice_client(self):
stop(self):
clear_queue(self):
get_queue_info(self):
queue_length(self):
play(self, query: str, ctx: Context):
_process_song_queue(self):
_after_playing(self, error: Optional[Exception], finished: asyncio.
1
1
0..*
isinstanceof
BgDownloadSongQueue
_waiting_queries:
_downloaded_songs:
_processing_task:
_now_processing:
BgDownloadSongQueue.TrackedAsyncQueue:
__init__(self, song_downloader: SongDownloader):
next(self):
add(self, query: str, ctx: Context):
clear_queue(self):
get_queue_info(self):
queue_length(self):
_process_queue(self):
<<use>>
1
MusicCog
_bot:
__init__(self, bot: commands.Bot):
play(self, ctx: commands.Context, *, search: str):
skip(self, ctx: commands.Context):
stop(self, ctx: commands.Context):
pause(self, ctx: commands.Context):
resume(self, ctx: commands.Context):
loop(self, ctx: commands.Context):
queue(self, ctx: commands.Context):
clear(self, ctx: commands.Context):
_stop_music_player(self, guild_id: int):
_is_on_same_channel(ctx: commands.Context):
on_voice_state_update(self, member: discord.Member, before: discord.VoiceState, after: discord.VoiceState):
check_listeners(self):
monitor_music_player_status(self):
connect_on_command(self, ctx: commands.Context):
ensure_bot_on_voice(self, ctx: commands.Context):
1
1
<<use>>
SongDownloader
_yt_dlp_opts:
_youtube_regex:
SongDownloader.NoResultsFoundException:
SongDownloader.LiveFoundException:
SongDownloader.AgeRestrictedException:
SongDownloader.PlaylistFoundException:
__init__(self, song_cache: SongsCache, cookies_path: Optional[
prepare_song(self, query: str):
_load_cookies(self, cookies_path: Path):
_construct_song(self, query: str):
_get_url(self, query: str):
<<creates>>
Song
title:
url:
duration:
thumbnail:
expires_at:
_stream_url:
_ffmpeg_options:
get_source(self):
<<use>>
YtDlpLogger
_LOG_PREFIX:
debug(self, msg):
info(self, msg):
warning(self, msg):
error(self, msg):
1
1
SongsCache
__contains__(self, query: str):
__getitem__(self, query: str):
__setitem__(self, query: str, song: Song):
isinstanceof
<<use>>
LRUSongsCache
_youtube_regex:
__init__(self, songs_size: int, queries_size: int = 500):
__contains__(self, key: str):
__getitem__(self, key: str):
__setitem__(self, query: str, song: Song):
LRUCache
1
2
<API>
Discord API

Figure 1: UML Class Diagram of the Discord Music Bot

Key Features

The main feature of the Discord Music Bot is its ability to play music by simply entering a URL or a query for YouTube songs. The bot uses yt-dlp to stream songs directly from YouTube. When a user requests a song, it is added to the playback queue, and the bot downloads the songs in the background to ensure that they are ready to play on demand. To enhance performance and minimize redundant downloads, the bot also implements caching for previously downloaded songs.

One of the key advantages of this bot is its ability to play music simultaneously across multiple Discord servers. Each server operates independently, allowing users to enjoy seamless music playback in different servers at the same time without interference. This feature makes the bot highly scalable and suitable for large communities.

The bot offers a variety of commands that enable users to control the music flow, manage the queue, and customize playback settings. For a comprehensive list of all available commands and their usage, users can enter the *help* command directly in Discord or refer to the documentation available in the GitHub repository. These features ensure that users have full control over the music experience.

Testing and Evaluation

The process was carried out manually by creating a dedicated Test Server on Discord. This approach allowed for the verification of all functionalities in a controlled environment, ensuring that each command performed as expected. During the testing process, various commands were executed in real-time to assess the bot's behavior, including music playback, queue management, and user interaction.

Additionally, logging commands in debug mode proved to be an invaluable tool for tracking the bot's actions and identifying any potential issues. By logging detailed command execution data, I was able to closely monitor the flow of operations, detect any errors or inconsistencies, and refine the bot's performance. This process allowed for systematic troubleshooting and optimization of the bot's functionality.

Deployment

The Discord Music Bot is deployed using Docker, ensuring a consistent and portable runtime environment across various platforms. Docker simplifies the deployment process by packaging the bot along with all its dependencies, including Python, ffmpeg, and required libraries, into a single containerized application.

The bot is hosted on Google Cloud Platform (GCP), which provides the scalability and reliability needed to handle multiple users and servers. Once the Docker image is built, it is pushed to GCP, where it runs continuously as a managed service. This setup ensures that the bot remains online and can efficiently process requests from multiple Discord servers.

Adding the bot to a Discord server is straightforward: users can use an invite link generated by the bot owner. This link allows server administrators to seamlessly integrate the bot into their communities, providing an always-available music playback solution.

Feature Improvements

The Discord Music Bot is designed to evolve and adapt to users' needs, with several planned feature improvements to enhance its functionality and user experience. One of the key improvements under consideration is the ability for users to create personal playlists. This feature would allow users to save their favorite songs and easily play them at any time, without the need to manually add songs to the queue each time. By creating and managing playlists, users can enjoy a more personalized music experience, improving the bot's usability and convenience for long-term listening.

Additionally, the bot's modular architecture, based on the use of Cogs, enables easy integration of new features. One such cog that could be added in the future is *TeamSplitter*. This cog would provide functionality for dividing users from a voice channel into separate teams for gaming or collaborative activities. It would automatically move users into different team channels, facilitating better organization and communication during multiplayer gaming sessions or other group activities. This would make the bot even more versatile, serving not only as a music playback tool but also as a useful addition for gaming communities and events.

Conclusion

The Discord Music Bot provides a robust and user-friendly solution for music playback in Discord servers, offering intuitive controls for managing queues and enhancing the listening experience. Its modular architecture, built with Cogs, ensures flexibility and facilitates future enhancements. Deployed reliably using Docker on Google Cloud Platform, the bot is scalable, accessible, and optimized for diverse server environments.