Contents

[**1. Problem Definition** 2](#_Toc498951541)

[**2. Economic Feasibility Study** 2](#_Toc498951542)

[**3. Software Requirements specification** 3](#_Toc498951543)

[Functional requirements 3](#_Toc498951544)

[Use cases 4](#_Toc498951545)

[Web GUI: 4](#_Toc498951546)

[Message Board: 4](#_Toc498951547)

[Software Qualities 5](#_Toc498951548)

[**4. Design Specification** 5](#_Toc498951549)

[Software Architecture 5](#_Toc498951550)

[Design Patterns 7](#_Toc498951551)

[Class Diagrams 8](#_Toc498951552)

[Pros of Draw.io: 14](#_Toc498951553)

[Cons of Draw.io: 14](#_Toc498951554)

[**5. Programs** 15](#_Toc498951555)

[Component Diagrams 15](#_Toc498951556)

[Deployment Diagram 19](#_Toc498951557)

[List of Classes & functions: 19](#_Toc498951558)

[Total # of classes: 50 22](#_Toc498951559)

[Link to the web based application: 22](#_Toc498951560)

[**6. Technical Documentation** 22](#_Toc498951561)

[Languages: 22](#_Toc498951562)

[Reused Programs: 22](#_Toc498951563)

[Software Tools and Environments: 23](#_Toc498951564)

[**7. Testing** 23](#_Toc498951565)

[Application Testing 23](#_Toc498951566)

[Robustness Testing 24](#_Toc498951567)

[User Interface Testing 25](#_Toc498951568)

[Time-Efficiency Testing 25](#_Toc498951569)

[Contributions: 25](#_Toc498951570)

# **1. Problem Definition**

Discord is a new VOIP and messaging service that went live 2 years ago, and over that time has improved in capabilities and openness to developers to provide a working interface for their applications to work alongside the Discord application. As such there are core issues with any scalable application causing server costs to increase at a faster rate when linear scale of users, and how a simple to complex web GUI to interface and help manage the bot through other means other than command line through the messaging system. Improving the qualities of existing modular Discord bot services to provide a seamless integration of new features without impeding performance and the efficiency of the server which is hosting the Discord bot client. This requirement to improve scalability so that multiple users can be on and still use the multiple features that the bot will be able to use will require extensive usage of minimizing network constraints and hardware constraints by threaded optimization. Building and managing a core platform that acts as the central point of communication for the modular components and provide a simple basis to interact with the Discord API. Utilizing a simple server client communication between the hosting hardware and the client to enable a command line operations as well as having a web GUI to manage the bot.

# **2. Economic Feasibility Study**

               Discord is a relatively new VOIP (voice over internet protocol) service that is geared towards gamers (persons that play video games for leisure or for other reasons) and software developers. This is a service that is to replace the usage of skype and other VOIP services such as TeamSpeak, Mumble and etc. As a service built for those in mind, they have developed a way for developers to interact with their application through RESTful API. As such many 3rd party applications were able to be made to help people manage their Discord Servers.

There is a very small subset of Discord application that have a way to manage the server through other means other than hosting the application on your own computer or server where you can configure the bot internally or through the Discord messaging system that utilizes command line like syntax. With this application we are able to develop a system where a bot is configurable from those same avenues but utilizes a web GUI to help server owners and users to look through statistics and manage the bot through a GUI. As such this can make it possible for server owners to host this application on their own server such as a home computer but allows data to be sent to update our database so that there is a global setting that each bot is set to do

              The software to be built uses a C# wrapper that interacts with the Discord API, called DSharp Plus made by a user called “NamloosDT”, that transcodes json objects for the Discord API to interact with using C#. This source code is open source, so this allows the software to inherit the constraints from the Discord application

           The application developed is being built in three different parts, the core engine that utilizes the wrapper and a custom-built API. The custom-built API will act as an interface for the applications built for this engine to interact with DSharpPlus as well as the database if given access using a dictionary (key and value) type of database to enable a dynamic way to add new features to the database without changing the core aspects of the database while still adhering to new components database requirements. The web GUI will be built upon the Material UI design templates by Google to allow a easy way for the site to be dynamic and simply present information and interactive components to the bot. The web GUI provides at its core an overview dashboard experience to interact with the bot of your server, currently limited to this bot only until the API provides way for other bots to utilize. The third and final part is the console portion of the bot which allows the user to host the bot on their own accord such as their own server or computer, simply put this will allow our central server to serve more people while marginalizing cost.

    The bot being hosted on the user’s own server/computer provides a leeway in providing a more sustainable model for hosting several servers at once as the cost of hosting will be offsetted to the user if they choose to do so, however this still allows the us, the developers to host the bot as well to serve users that do not want to host the bot on their server. The bot will run on linux or windows operating systems so to reduce costs in server hosts as linux is a cheaper alternative to windows servers, while still serving majority of the computer users prefered operating system. If the user hosts a bot on their own computer, it will allow them to view real time data debugging if they feel that errors are popping up to help them figure out issues quickly. The user will also be able to quickly add new modules or delete them using just their filing system.

           The application being developed is to bring forth a way to manage your Discord server through multiple different ways, namely two that can be done within a reasonable amount of time, through an MVC and through the Discord messaging system. The MVC site contains a way to look at statistics pertaining to their server which they are part of and manage the server through a web GUI. The web GUI is built on the concept of single page application with dynamic sizing to enable porting it to mobile devices.

# **3. Software Requirements specification**

## Functional requirements

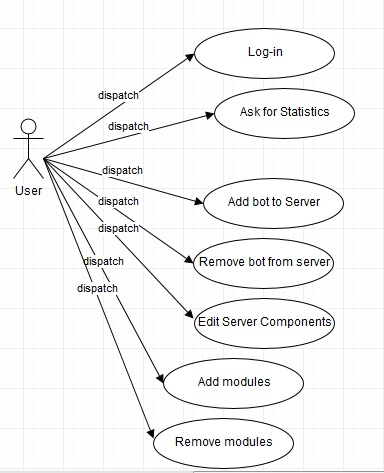
User: can see/use the from discord application, any modules the bot has and any bot commands that aren’t role exclusive. Multiple users would be able to give command at once, so the bot need to be prepared for that.

Server Admin: Can see the user’s views as well as the views of the Web GUI, which include the statistics of the bot. Able to Log-in, connect or disconnect mods from the bot, and add or remove a bot from a server, as well as edit that bot’s components on the server. Needs to overrule commands by users.

Developer: can see all views that a user or Server admin can see, as well as having access to some of the testing method. [The developer has access to, and can make additional code for the bot, usually in the form of more modules.] Can only see Web GUI data from the database, because it requires a discord account and the developer would not necessarily have the admin’s.

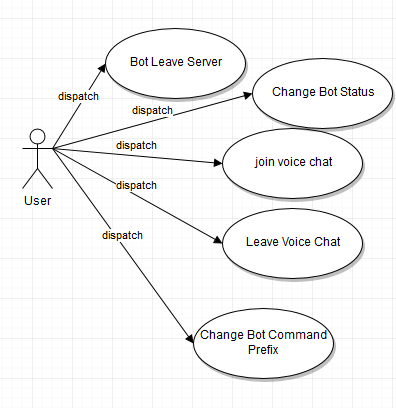
## Use cases

### Web GUI:



From the Web GUI, the user can log-in and authenticate using discord, see information within their scope as a user, seeing servers they can interact with, add or remove the bot these severs, edit the configuration of the connected servers, and check modules available to the bot.

### Message Board:



From the message board, the user interact with the bot using a prefix as long as they have sufficient permission.

## Software Qualities

-Correctness: A given requirement, the project needs to work as well as possible without errors.

-Scalability: The project is meant to potentially be used by many people in many ways, likely far more than just what it can do at the time of deployment, so it needs to be able to handle all of that. This program is meant to be used on a widely used service. As such, it needs to be able to handle a potentially tremendous user base. With the uses of multithreading requires a need to implement locks or semaphores to avoid file locking and starvation between 2 threads.

-modularity: The nature of the program is to be able to fit new functions on to it as a user desires. Modules can be enabled at the users request and the system is able to integrate that seamlessly.

-efficiency: The program is expected to be fast due to the way discord works often slowing down programs. It doesn’t need anything more working against its speed where possible. Multithreading processes makes the bot able to diversify workloads, and the code for this is in program.cs, where new threads are made. All Modules work from separate threads. The music player, for example, is on a separate thread from the main thread, communicating only when ending music. Asynchronous managements makes it possible to balance these processes out while maintaining the main code running without blocks

-robustness: A must for any program, this project must be able to handle errors and incorrect inputs to prevent any critical errors from ruining the user experience. As a multi-threaded system, the program allows for errors to be caught and dealt with by ignoring the issue and logging it while avoiding blocking the main thread and shutting the system down.

-Security: The system can go up to 2 factor authentication because it uses Discords OAuth2 system to authenticate users, and so only works for discard users. Hides a lot of the information to public systems such as the database and internal credentials in secret managers in visual studios.

-maintainability: The system needs to be updated to be more convenient, and not outdone by other bots, otherwise it does not have much reason to exist. This is provided by the way a lot of the components have external coupling to provide an easy way to take out components and refactor them if needed, without changing components that use it. This is the repository pattern for building the database. Also, everything is interfaced utilizing the DSharpPlus wrapper, allowing us to create modules and reducing high coupling that makes the project unable to expend.

-user friendliness: If a user is expected to use a program, the program should be expected to be navigated by that user, and this program is no different. We just want it easy to read and navigate to frequently used commands and functions.

# **4. Design Specification**

## Software Architecture

This project used a 3 tier system for the Web GUI, the tired and tested view, business, and controller layers. The bot utilize 3 interfaces, Web GUI, Discord Client, & Database. In comparison to the Web GUI, the bot itself uses 2 layers, a view and controller layer in one in the discord client, and itself as a business layer. See fig. 1 to see the bot take the model and map it to the database using a repository pattern and using that save any information being changed. It’s simple for the bot to save changes to the model of the database and the component that deals with synching changes to and from the database with ease using object templating. For convenience, the bot and the Web GUI both connect to the same database. The project is a multi-threaded application, which allows it to efficiently do multiple tasks at once without endangering the bot’s core with potential system crashes. This is done by having the CoreDiscord class create asynchronous threads for other applications, allowing all of them to work independently. See fig. 2.

Fig 1.



Fig.2



## Design Patterns

**RESPOSITORY PATTERN**

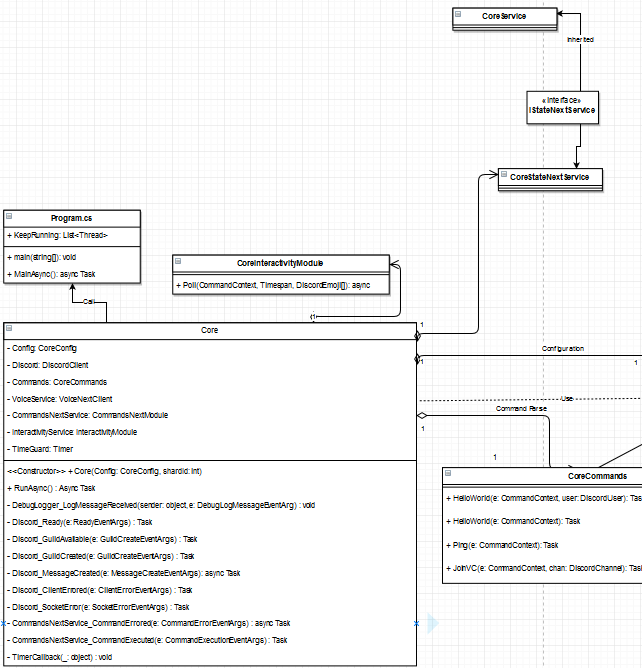
One of the design patterns used for this bot is the Repository pattern, which is utilized in the database interface. The pattern uses EntityFrameworksCore to create a way to take a class and have it act as a model for repositories. Then it uses Generic Type Parameter to handle database context builds and the actual simplifies quarying mechanism by forcing it through a template and having the compiler decide how to implement the SQL demands, with context acting as the deciding factor. Context Build maps to SQL commands such as DELETE, SELECT, ADD, and UPDATE, among others. Both synchronous and asynchronous methods were implemented to properly handle syncing, and the data layers are isolated to reduce redundant columns. The models provide entities that are strongly typed, and thus behaviours can be debugged easily when it comes to data type errors. Another one of the benefits of this pattern is that it is easy to map models to columns in the database; It is also able to build tables when not available using the EntityFrameworks code and utilizing the models built using that repository pattern.

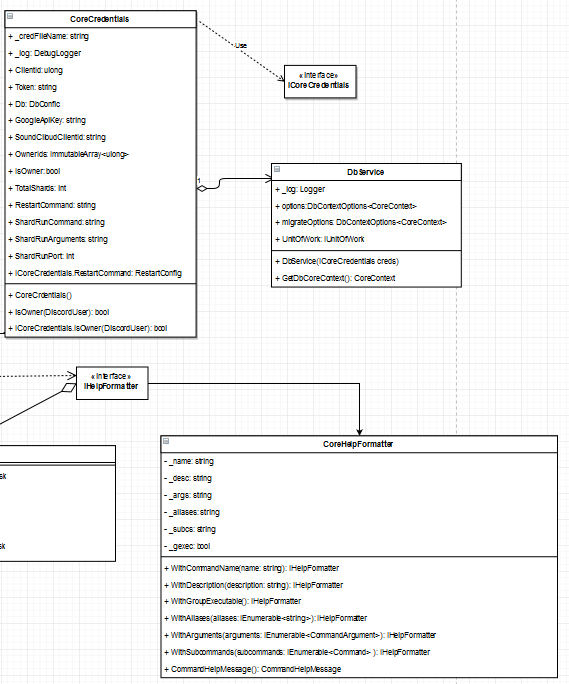
**WRAPPER PATTERN**

The other design pattern used is the C# wrapper, which is used as a base model for the whole entire project. It provides access to the Discord API using C#, allowing for code written by the developers to use the returned data from the API. This is used to communicate between the Discord client and the bot. DSharpPlus is the wrapper used, which formats the API into strongly typed C# classes and easily accessible and usable components. Allows for flexible use of the classes and integration into the design of the code, and from there, translation into implementation.

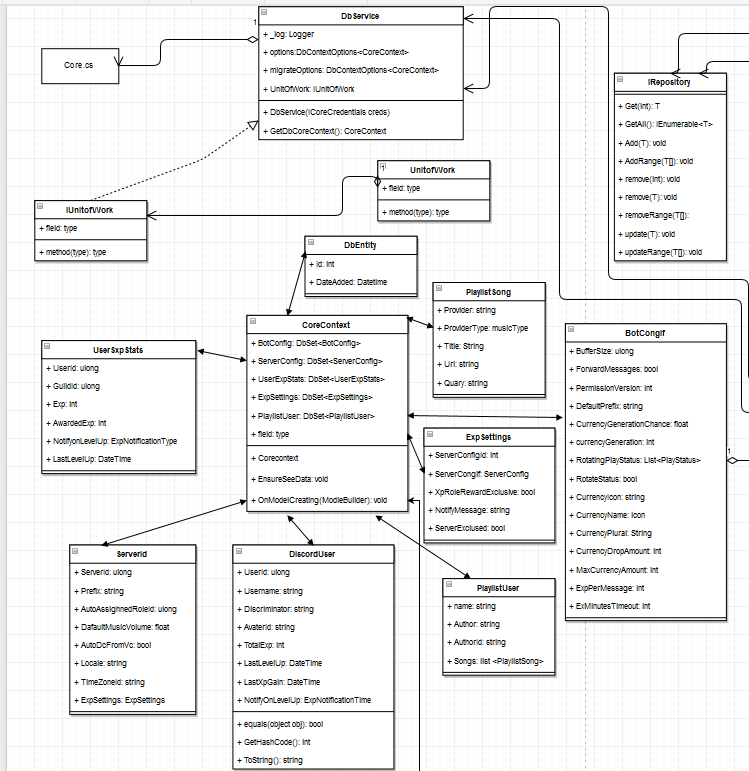
## Class Diagrams

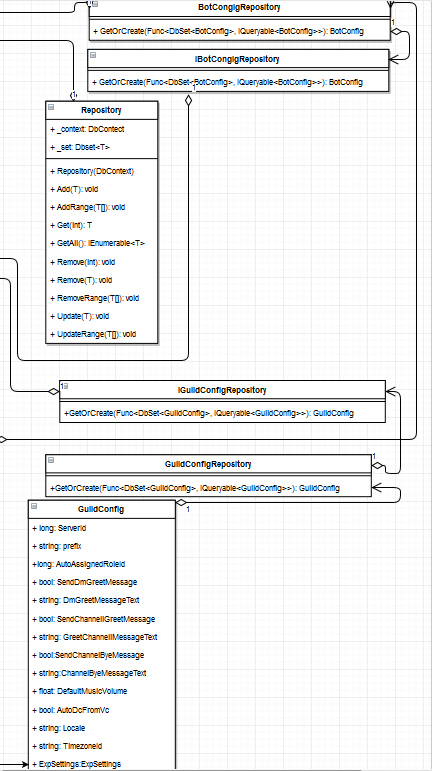
Core:



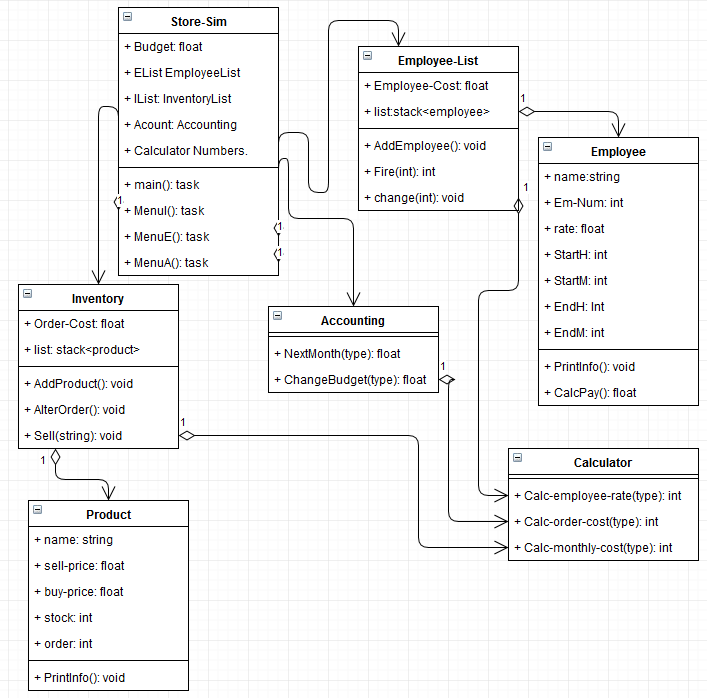


CoreDatabase:

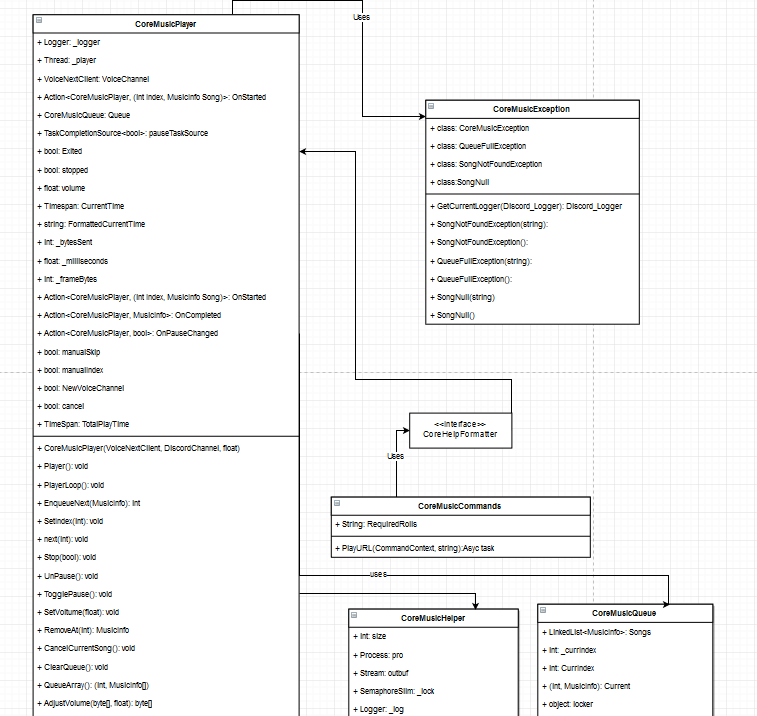


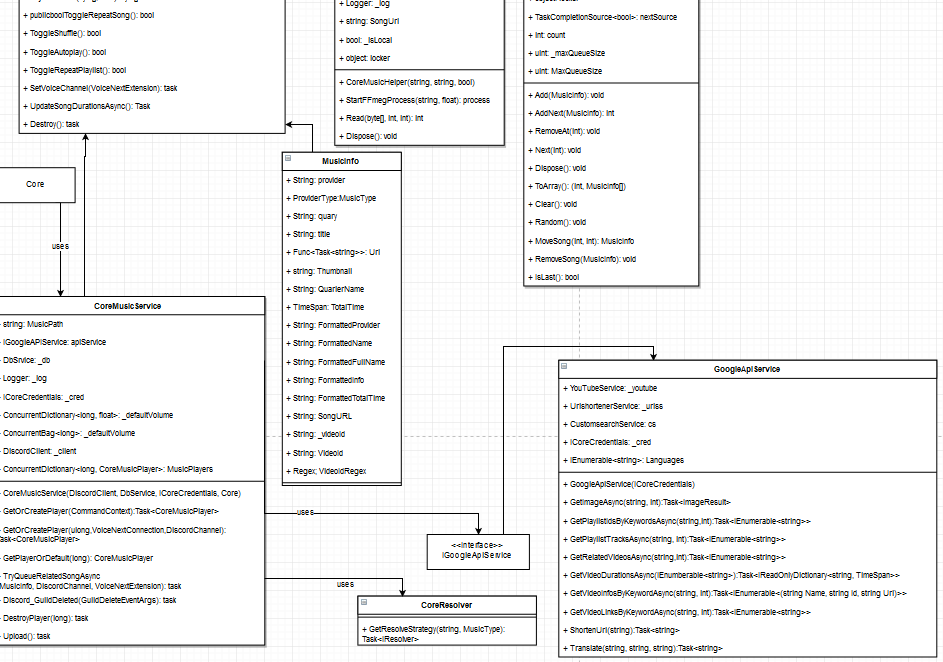


StoreSim Module:



Music-Module:





## Pros of Draw.io:

-Draw.io is web based tool that requires no downloading or signing up, meaning anyone with internet access can use it with no hassles or risks of any kind.

-The service works with GitHub, allowing for a user to access any of their projects and any .xml files in those projects easily, while saving them right back to the repository.

-It can also access files from things like Dropbox or Google Drive, though we have no experience with that.

-When in use, Draw.io allows the user access to nearly any UML entity they could want, as well as any connections.

-Diagram creation itself is very easy to understand; one just needs to drag entities and edges into the graph, fit them together, and name them, which is all immediately understandable.

-In addition to the detailed connections in the side bar with the entities, each entity can also make basic connections leading to other entities that are less detailed. They are also angle based instead of a maneuverable straight line like the side bar connections

-These connections can also snap into place on the entities, tracking them if they are moved.

-Draw.io features a sketch pad that lets the user place a fully written object or group of objects on to it, and then drags them out like the blank entities. The benefits of this are that these objects are saved on the sketch pad in future diagrams, and the sketch pad can hold quite a number of them.

-The ‘to front’ and ‘to back’ commands allow for easy manipulation of layers.

-Along with UML, the service is also equipped to make charts and diagrams of other kinds like flow charts or entity relation diagrams.

## Cons of Draw.io:

-Viewing other group member’s work can be a hassle, as you can’t just open a .xml file to view it, you have to download it and open it in Draw.io.

-It can also be trouble to figure out because the only form of tutorial is a help bar connected to a search engine for a help site, which is not too helpful if the user does not know what to search for.

-Changing the text on an object without getting rid of everything already there, for example, is tricky if the user doesn’t now to press enter first.

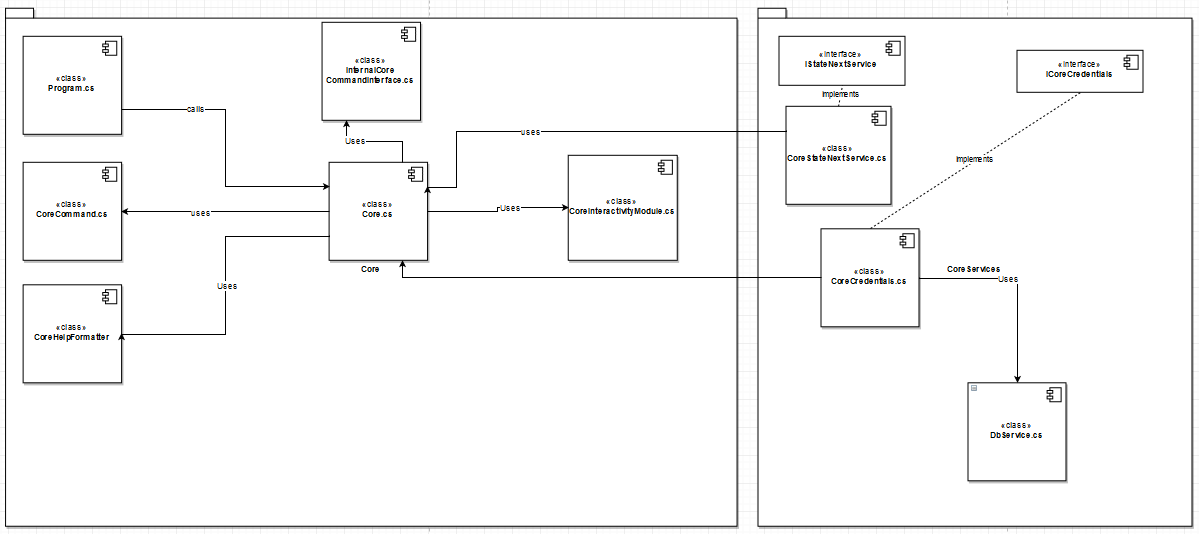
-Copy and pasting an object or group of objects is difficult, as you can only save them to the sketchpad on the diagram that had them, refreshing the receiving diagram, and then placing the object there.

-Internal sealed classes and C# encapsulations could not be represented using the UML tools available.

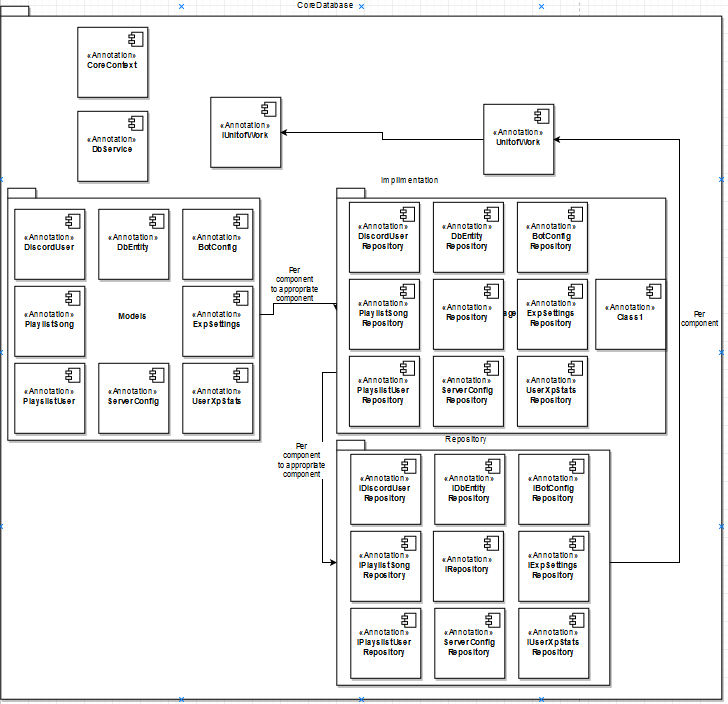
# **5. Programs**

## Component Diagrams

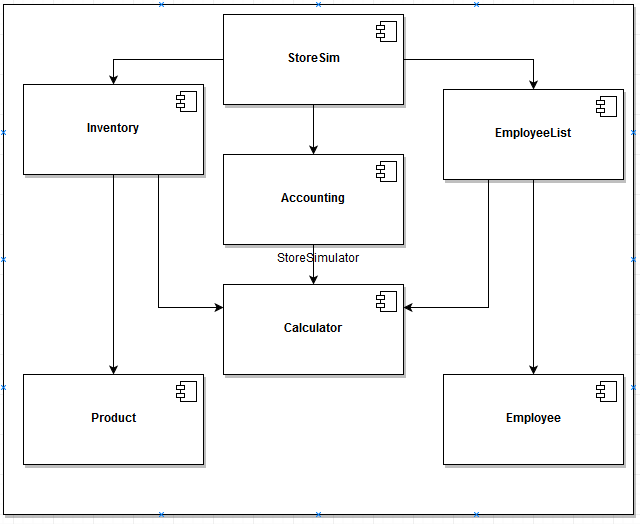
Core:



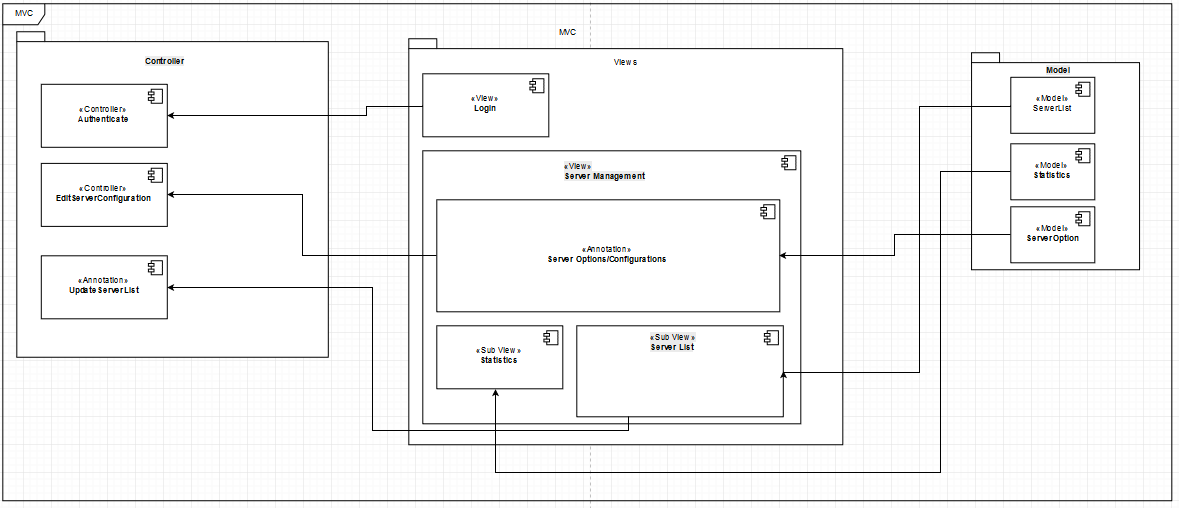
Database:



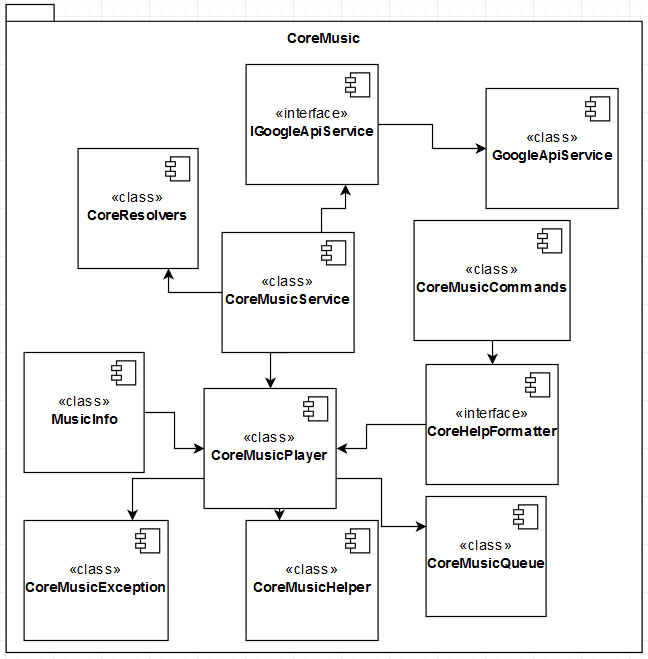
StoreSim:



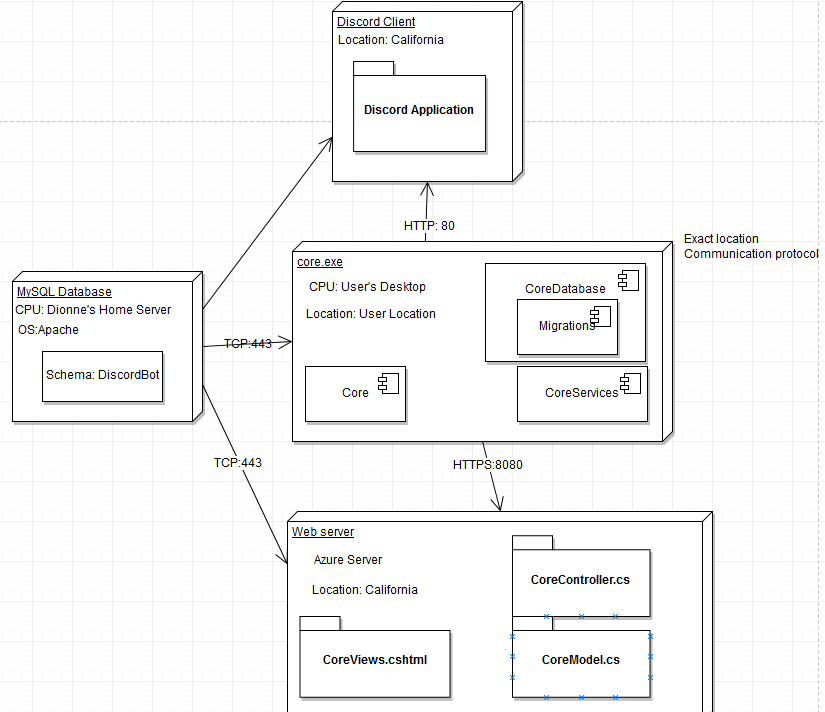
MVC Model:



Music Module:



## Deployment Diagram



## List of Classes & functions:

|  |  |
| --- | --- |
| Classes: | Functions: |
| **Core** | **+Core**  **+RunAsync**  **+DebugLogger\_LogMessageRecieved**  **+Discord\_Ready**  **+Discord\_GuildAvailable**  **+Discord\_GuildCreated**  **+Discord\_MessageCreated**  **+Discord\_ClientErrored**  **+Discord\_SocketError**  **+CommandNextService\_CommandErrored**  **+CommandNextService\_CommandExecuted**  **+TimerCallback** |
| **Program** | **+main**  **+MainAsync** |
| **CoreInteractivityMedule** | **+Poll** |
| **CoreCommands** | **+HelloWorld(e: CommandContext, user: DiscordUser)**  **+HellowWorld(e: CommandContext)**  **+Ping**  **+JoinVC** |
| **CoreHelpFormatter** | **+WithCommandName**  **+WithDescription**  **+WithGroupExecutable**  **+WithAliases**  **+WithArguments**  **+WithSubcommands**  **+CommandHelpMessage** |
| **CoreCredentials** | **+CoreCredentials**  **+IsOwner**  **+ICoreCredentials.IsOwner** |
| **DbService** | **+DbService**  **+GetDbCoreContext**  **+UnitOfWork** |
| **CoreService** |  |
| **CoreStateNextService** |  |
| **CoreService** | **+GetUpTime**  **+GetUpTimeString**  **+Initialize**  **+Print** |
| **CoreContext** | **+Corecontext**  **+EnsureSeeData**  **+OnmodelCreating** |
| **DbEntity** |  |
| **PlayListSong** |  |
| **BotConfig** |  |
| **ExpSettings** |  |
| **PlaylistUser** |  |
| **DiscordUser** | **+ equals**  **+ GetHashCode**  **+ToString** |
| **ServerId** |  |
| **UserExpStats** |  |
| **Repository** | **+Repository**  **+GetAll**  **+Get**  **+Add**  **+AddRange**  **+remove(int)**  **+remove(T)**  **+removeRange**  **+update**  **+updateRange** |
| **BotConfigRepository** | **+GetOrCreate** |
| **IRepository** | **+Get**  **+GetAll**  **+Add**  **+AddRange**  **+remove(int)**  **+remove(T)**  **+removeRange**  **+update**  **+updateRange** |
| **IBotConfig** | **+GetOrCreate** |
| **DbService** | **+DbService**  **+GetDbCoreContext** |
| **UnitofWork** | **+UnitofWork**  **+Disposed**  **+Dispose** |
| **CoreMusicPlayer** | **+CoreMusicPlayer** |
| **MusicInfo** |  |
| **MusicCommands** | **+PlayURL** |
| **CoreMusicException** | **+GetCurrentLogger**  **+SongNotFound()**  **+SongNotFoundException(string)**  **+QueueFullException()**  **+QueueFullException(string)**  **+SongNull()**  **+SongNull(string)** |
| **CoreMusicHelper** | **+CoreMusicHelper**  **+StartFFmegProcess**  **+Read**  **+Dispose** |
| **CoreMusicQueue** | **+Add**  **+AddNext**  **+RemoveAt**  **+Next**  **+Dispose**  **+ToArray**  **+Clear**  **+Random**  **+MoveSong**  **+RemoveSong**  **+IsList** |
| **CoreMusicService** | **+CoreMusicService**  **+Upload** |
| **GoogleApiService** | **+GoogleApiService**  **+GetImageAsync**  **+ GetPlaylistIdsByKeywordsAsync**  **+GetPlaylistTracksAsync**  **+GetRelatedVideosAsync**  **+GetVideoDurationsAsync**  **+ GetVideoLinksByKeywordAsync**  **+GetVideoLinksByKeywordAsync**  **+ShortenUrl**  **+Translate** |
| **StoreSim** | **+main**  **+MenuI**  **+MenuE**  **+MenuA** |
| **Inventory** | **+AddProduct**  **+AlterOrder**  **+Sell** |
| **Product** | **+Printinfo** |
| **EmployeeList** | **+AddEmployee**  **+Fire**  **+change** |
| **Employee** | **+Printinfo**  **+CalcPay** |
| **Accounting** | **+NextMonth**  **+ChangeBudget** |
| **Calculator** | **+CalcMonthlyCost**  **+CalcOrderCost**  **+CalcEmployeeRate** |

## Total # of classes: 50

## Link to the web based application:

<https://discordapp.com/oauth2/authorize?&client_id=362345589561622539%60E&scope=bot&permissions=0>

This is an invite link, which will add CoreDiscord to the user’s current Discord server.

# **6. Technical Documentation**

## Languages:

C#, HTML5, JavaScript, MySQL, CSS

## Reused Programs:

-Store Project from Matthew’s previous work, now rewritten for C# and Discord.

-Discord API from Discord

-C# Wrapper from Discord

-testing components for commands from Dsharpplus.Test

-NadekoBot - Examples in layering services

-DsharpPlus.Test for examples in using Dependancy Injections

-MusicBot by Chrisradio and Nadeko.MusicPlayer for music module

- Authentication uses passport-discord

- database sync changes NadekoBot

## Software Tools and Environments:

-Visual Studios: used in the core engine. It’s an instanced class we can call from.

- .Net 4.7: used with the wrapper to make it work. .Net 4.7 is used to get access to AspConsoleCore and EntityFrameworkCore frameworks and have access to newly developed libraries that are only available on .Net 4.7

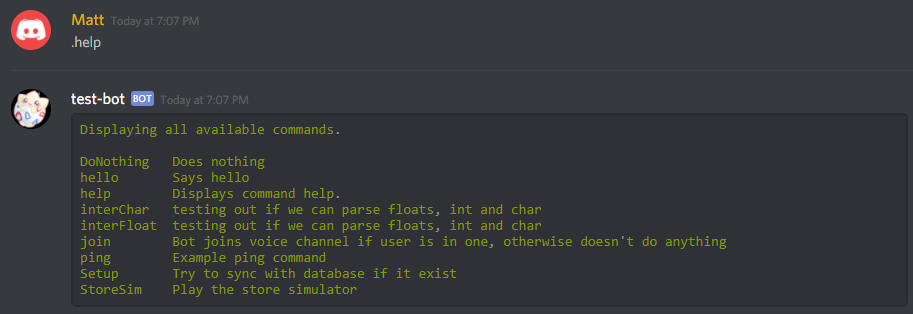
- Visual Studio 2017 Community used for compiling and writing majority of the bot. Usage of Intelligence saves a lot of time when it comes to implementing interfaces and searching for available methods/variables within scope.

- reactjs/nodejs used to build a single page website. Utilizes angular to route all those pages so a single page view is seamless

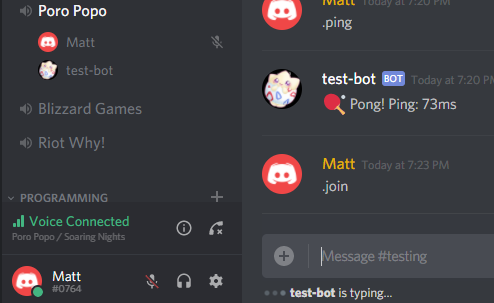
# **7. Testing**

## Application Testing

-Case: use .help to display options.



-Case: use .join to get the bot into a voice chat.



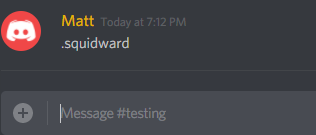
-Case: Download 1 song on Music Mod.

-Case: While using .StoreSim user goes to, and then exits from, each of the 3 sub-menus.

-Case: Log in using Discord account on the Web GUI.

## Robustness Testing

-Case: User enter a nonexistent command without using a mod.



Result: no response from bot

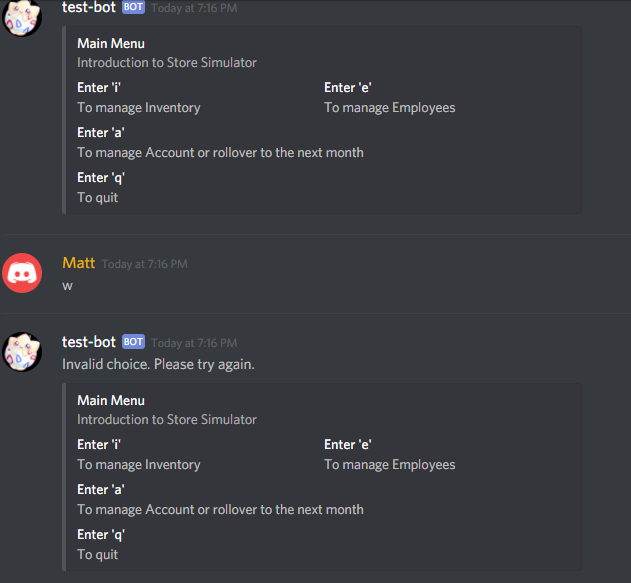
-Case: use .join to get the bot into a voice chat without being in a voice chat yourself.



Result: Bot looks to see if user is in a voice chat, and then makes to further response.

-Case: User try to play a song with the music mod that does not exist.

-Case: User inputs an incorrect command at a menu.



-Case: The user attempt to log in to the Web GUI using nonexistent account.

## User Interface Testing

-

-

-

-

-

## Time-Efficiency Testing

# Contributions:

Dionne:

Documentation:

+ Designed and planned project

+ Helped and guided Matthew for the class and component diagrams

+ Deployment Diagram

+Economic Feasibility Study

+ Problem Definition Problem

+ Software Architecture - Matthew helped with formatting and ensuring it was comprehensive

Programming:

+ Built the Core Engine

+Built Music Module

+Preliminary Commands for Core and Music Module

+Built Services

+Repository Database

+Syncing Database

+Models for all classes

Matthew:

+ Creation of Use Case Diagrams

+ Write up of functional requirements.

+ Write up of Software quality list.

+ Write up of Software Architecture explanation.

+ Write up of design pattern explanation.

+ Creation of Class and Component Diagrams.

+ Creation of deployment diagram.

+ Write up of technical documentation.

Note: all of the above work was done with direction from the team leader Dionne, or fallowing the layout of Dionne’s code

+ Creation of the Class & function list, as well as counting up the total number of classes.

+ Write up of the Pros & Cons of Draw.io.

+ repurposing of StoreSim code to work in C#, and usable in Discord by the bot.

Pranesh: