System for configuring Discord bots via a web interface

Ioan Clarke

MSc Computing

School of Computer Science and Informatics
CARDIFF UNIVERSITY

Supervisor:
LIAM TURNER

November 5, 2021

1 Abstract

Bots allow a wide range of additional functionality to be integrated into Discord servers; from playing music and posting memes, to automatic moderation and providing functioning economies (Top.gg 2021b). Often, bots allow configuration of some parameters; for example, the message to send when a new user joins the server, or the nickname of the bot.

There are two main problems with the current way in which bots are configured. Firstly, most configurable bots are configured via users sending specific messages that are recognized as ‘commands’ by the bot; the bot can then adjust its behaviour according to the command that was sent. This may be manageable when there is only a small number of parameters that can be configured, but as the number of parameters or the number of configurable bots grows, problems arise, as configuring the bots relies on remembering a potentially large range of commands for each of them. Secondly, making a Discord bot configurable is very complex. Suppose a developer has created a Discord bot. If they want users to be able to configure the bot, the developer must create a database to store configurations, an API that allows the bot to communicate with the database, and if they want the bot to be configurable via a graphical interface, a front-end GUI.

To solve these problems, a system was created to allow Discord server administrators to configure bots in their servers using an online graphical interface. The system is such that developers can easily allow their bots to be used with it. This allows server administrators to conveniently configure bots and greatly reduces work for bot developers who wish for their bots to be configurable.

2 Acknowledgements

I would like to thank my project supervisor, Liam Turner, for his excellent support provided throughout the project.

Also, thanks to Kenrick Mock of the of the University of Alaska Anchorage for providing me with valuable sources of information regarding Discord.

Thank you to my friend Theo for his inspiration regarding creative ideas for Discord bots.

Thanks to my parents for their help throughout all of my education.

Finally, thank you to my lovely girlfriend Charlie for her moral support during the project.

Contents

List of Figures

3 Introduction

Discord is an application that provides free voice, video, and text chat within communities called ‘Servers’. It is available for most popular desktop and mobile operating systems (Discord 2021a). Within these servers, conversations are separated into separate ‘channels’. For example, in a server for an MSc Computing course, there may be a channel for each module on the course. This helps keep conversations that are related to different topics separated and categorised.

Although Discord was originally designed to be a platform for gamers to communicate, it has since found use in many areas, including education (Vladoiu and Constantinescu 2020) and hobbyist groups (West 2020). These will be explored in the Background Material chapter.

Discord incorporates a feature called ‘bots’, which are effectively Discord clients that are not directly controlled by a human. Instead, a bot is a pre-programmed client that responds to specific messages that are sent in servers that the bot is present in, based on how the developer configured the bot. The messages that the bot responds to are known as its ‘commands’. Bots are generally used to automate tasks and provide extra functionality to a Discord server, similar to how bots are used on other platforms such as Messenger (Facebook 2021) and Teams (surbhigupta et al. 2021).

There are bots which allow configuration of certain parameters, for example changing the welcome message when a new user joins the server, or altering the list of words that are forbidden in the server (Top.gg 2021c). Configuring bots in a Discord server is usually performed by sending specific commands to the bot. For example, a bot developer may program the bot to change the welcome message it sends to new members of the server by detecting when a message of the form ‘!welcome ⟨message⟩’ is sent and then changing the welcome message to ‘message’. So if an administrator of the server wanted to change the welcome message to ‘Hello!’, they would send the message ‘!welcome Hello!’ to any channel in the server.

As an administrator of a Discord server, configuring bots can be complex. Consider the situation where there are 20 or 30 configurable parameters, such as in the case of the popular bot, Dyno (Top.gg 2021c). Remembering the command to configure each of these these parameters is not practical, and so the traditional method of configuring bots by sending sending messages is flawed.

The proposed solution to this problem is to create a system for configuring bots that includes an online graphical interface that allows server administrators to configure bots in their servers in a more convenient way.

There is a fairly recent addition to Discord called 'Slash Commands'. Slash Commands allow a bot application to define a set of commands, which each include a name, description, and set of options similar to arguments provided to a regular function. They provide a less error-prone way of sending commands to a bot since they enforce a pre-defined structure (Discord 2021b). Slash Commands also provide an auto-complete feature, helping users easily see commands that are available. Therefore, Slash Commands offer major advantages over traditional bot commands. However, they are not a complete solution to the problem discussed above, since they still require the user to type many commands if they wish to configure many parameters of a bot. This is demonstrated by the fact that two very popular Discord bots, Arance and Dyno, support Slash Commands, but also have online graphical dashboards for configuration (Arcane 2021; Dyno 2021). This suggests that graphical interfaces are still a more convenient method of configuration than sending messages, implying that this functionality is desired and useful. However, the dashboards provided by these bots can only be used to configure their respective bots, and so a more general solution that can be used to configure any bot is still required.

Another primary benefit of the proposed system is that it will allow bot developers to easily incorporate configurability into their bots. Consider a developer who has created a bot and wishes to make it public, while allowing server administrators to configure certain parameters of the bot. This developer would need to create a database to store the configurations, an API to allow the bot to communicate with the database, and likely a front-end interface that allows server administrators to configure the bot - this is clearly a large amount of work. However, the system created during this project will mean that any bot developer can easily incorporate configurability into their bot by simply adapting their bot to work with the system. This is likely to drastically reduce work required by developers who wish to make their bots configurable.

Note: throughout this dissertation, the word 'guild' may be used interchangeably with 'server' since this is what Discord servers are referred to as in the Discord developer documentation and is the official terminology.

3.1 What follows

The following chapters in this dissertation will discuss the project and the system that is developed in detail. The chapters, and a brief description of each of them, are:

- Aims and Objectives - an outline of what the project aims to achieve, including some well-defined requirements.
- Background Material - an exploration of the literature surrounding Discord and bots, and existing solutions to the problem. It also justifies the project.
- Approach - an explanation of the intended approach to address the problem, including a justification of the approach, and why it was chosen over alternative approaches.
- Design and Implementation - an explanation of how the approach was implemented and the design of the project; this includes a justification for the chosen implementation.
- Evaluation - an assessment of the successes and failures of the project, including how it compares to the original requirements, as well as further work that has been identified.
- Conclusions - a summary of the successes, failures, and deliverables of the project, and an explanation of how the project contributes to the existing body of knowledge.
- Reflections - a reflection on what was learned by completing the project, including what areas have been identified as strengths and weaknesses, and what areas have been selected for continued personal development.

4 Aims and Objectives

This project aims to create a system that:

- Allows administrators of Discord servers to configure bots in their servers using an intuitive graphical user interface
- Allows Discord bot developers to easily allow their bots to become configurable using this interface

There will be two deliverables produced for consumption by end users during this project: a web interface for Discord server administrators to configure bots in their servers, and a framework that allows developers to integrate this functionality into their bots.

To achieve this, a front-end online interface will be built; as well as a database to store the configuration of bots, a framework that can be integrated into bots to allow use of the system, and a back-end API to provide allow the different parts of the system to communicate. Therefore, there will be a total of four deliverables produced during this project, though only two of them will be directly utilized by end users, the frontend and the bot framework; the other two will be operated by myself.

Furthermore, the web interface developed for use by server administrators will aim to be as intuitive as possible, in order to make using the interface more convenient than configuring bots in the traditional manner of typing commands.

Also, the framework that will be built to allow bots to be usable with the system will aim to be as simple as possible, to encourage developers to integrate this functionality into their bots.

4.1 Requirements

Based on what was discussed above, there are several well-defined requirements for the system:

1. The system must allow server administrators to intuitively configure supported bots using the web interface
2. The system must allow developers to integrate the functionality into their bots in an uncomplicated manner
3. The system must be secure, to prevent unauthorized alterations to bots in the database

5 Background Material

5.1 Methodology

To perform the research required for this project, many sources relating to Discord, bots, and Discord bots were explored. Bots have a rich history so there is ample literature available that discusses them. However, Discord is a relatively modern technology, meaning the related literature is fairly limited. Furthermore, the literature surrounding Discord bots specifically is even more scarce, since it appears to be a very niche topic.

Nevertheless, as much literature as possible was considered before beginning the project, to ensure that development of the system was well justified. I attempted to focus on academic research and formal literature, but as discussed above, this proved to be quite limited, and so some less academic literature was also analysed.

The literature discussed relating to Discord primarily focuses on its use in many different fields, which demonstrates its widespread use and popularity, which, in turn, suggests that creating a system that allows Discord users to use the platform more conveniently is justified. However, the ethical issues and potential problems with creating tools for Discord is also discussed and used as reason that care must be taken when creating the system.

The literature covered that discusses bots is of a similar vein, primarily focusing on the rising popularity and applications of bots, which serves as justification for making the use of bots more accessible. The potential for unethical use of bots is also discussed.

Regarding Discord bots specifically, a study investigating the number of Discord bots found to be present in servers is discussed, which shows further reasoning that the development of a system that makes the use Discord bots more convenient is justified.

5.2 Discord

Discord was originally designed to be a platform for gamers to communicate, but has since transformed into more than that; this is evidenced by the fact its description of itself has changed from an “All-in-one voice and text chat for gamers” (Discord 2016) to “a place where you can belong to a school club, a gaming group, or a worldwide art community” (Discord 2021a). Since its evolution into a more general platform, Discord has found use in many areas.

Vladoiu and Constantinescu (2020) discussed the efficacy of using Discord as a communication platform within the Computer Science programs at the UPG University of Ploiesti. Although the authors admitted they had not received any formal feedback at the time of publishing, which limits the credibility of the claims made, they stated that the first-year students studying Computer Science integrated into the community quicker than ever before, despite the restrictions due to COVID-19. The authors implied that the use of Discord in the department contributed to this success. The authors also discussed the bot that was created for use within the Discord server. The bot performed useful tasks such as attendance tracking, providing links to homework assignments, and streamlining the process of students taking exams. This displays displays the versatility of bots within a Discord server and demonstrates their usefulness.

Mock (2019) found similar success in using Discord within the Computer Science & Engineering Department at the University of Alaska Anchorage. In a survey conducted of platform users, 64% of respondents claimed they found tutoring via Discord either very effective or extremely effective, and 73% stated they found the Discord server either very helpful or extremely helpful as a student. The sample size of n=56 of the survey was relatively small, but the results still appear promising for using Discord as an education tool.

These cases of successful integrations of Discord into education suggest that Discord is likely to prove useful in an increasingly wider range of fields. Additionally, they show the popularity of Discord is likely to continue increasing and, especially due to the discussion of a successful Discord bot provided by Vladoiu and Constantinescu (2020), a system to more conveniently configure bots present in a server is likely to prove useful to server administrators.

West (2020) discussed how they used Discord for software and data skills classes, which were moved online and made more accessible to them during the COVID-19 lockdowns. The author described how Discord made the classes convenient by allowing attendees to easily navigate between different channels, stating “the user experience was better than in most other platforms I’ve used”. However, this is somewhat countered by the claim the author made that, for a user who is not accustomed to moving between different areas and windows within a platform, this may still be challenging. West further elaborates on how Discord facilitates convenient use by describing the pre-defined templates that Discord offers when creating a server, allowing creators of servers to choose templates such as Creators & Hobbies, Study Groups, and Friends & Family. This helps to make Discord more accessible to the general population, not just those who are familiar with technology. Once more, this demonstrates the growing accessibility and use of Discord, and served as justification for making it more convenient to use.

The increasing popularity of Discord is blatantly evident when reviewing the work of Curry (2020), where it can be seen that the number of registered Discord users increased by 1100%, from 25 million to 300 million, between 2016 and 2020. With 6.7 million active servers and 4 billion minutes of conversation happening on Discord each day in 2020 (Carey-Simos 2020), it is clear that Discord is an extremely prominent platform within technological society, and so deserves the attention of this project.

Although Discord can provide fantastic utility, it does present several problems. Jargon (2019) highlighted this in their article *The Dark Side of Discord, Your Teen's Favorite Chat App; Its private gaming communities can be like unsupervised playgrounds, full of racist memes, vulgar talk and bullying*, where they stated how some users of the service send racist memes and vulgar jokes. The primary issue here was how easy it was for children to access this obscene content; the author stated they were able to find pornographic content and Nazi-related memes within fifteen minutes of signing up to Discord. What the author claimed must be regarded with circumspection due to the information being presented in a newspaper, where author bias is possible. However, the anecdotes given by interviewees in the article offer some reinforcement to the opinion presented, such as a 13 year old boy who claimed "I'll be in these servers where people change their names to "ISIS" and make 9/11 jokes."

Since then, Jargon (2021) has written another article explaining how Discord is attempting to become a safer platform. For example, the platform now blocks users between the ages of 13 and 18 from accessing servers marked as "NSFW" (Not Safe For Work). The opinions in the article are corroborated by a representative of the Family Online Safety Institute who stated that Discord earns a "B+" for the improvements in safety that have been made to the platform.

Nevertheless, it is evident that Discord provides opportunities for abuse and hosts a range of content that many would regard as unethical, and so care must be taken when developing a system that aims to make Discord more convenient and accessible.

5.3 Bots

Traditionally, the word 'bot' referred to a type of 'chatbot', where a user would engage in conversation with a program that attempted to mimic human responses. The definition has since shifted towards a program that provides interaction with a user via a conversation-like interface (Lebeuf, Storey, and Zagalsky 2018).

Bots are used in a wide range of fields, typically to automate tasks for a person or group of people. Bots can be invited to a Discord server to add functionality, from posting memes and playing music, to auto-moderation and implementing economies (Top.gg 2021c).

The concept of bots has been around for a long time. As far back as 1966, bots have been created in hopes of passing the Turing test (Lebeuf, Storey, and Zagalsky 2018). Bots have come a long way since then; originally intended to convince users they were conversing with a real person, bots now have widespread integration into mainstream services such as Facebook (Facebook 2021), Teams (surbhigupta et al. 2021), and Discord.

Additionally, Lebeuf, Storey, and Zagalasky presented the idea of bots eventually replacing apps – this was backed up by statements from Murgia (2015), cited in Lebeuf, Storey, and Zagalsky (2018), who claimed Facebook stated they "aim to 'replace apps' one bot at a time".

Klopfenstein et al. (2017) took this idea further, proposing the definition of a "Botplication". The aim of Botplications is to provide a single application that provides the functionality of what would traditionally require many different applications. The authors did not present data on whether users desire this kind of service, but they did put forward several compelling arguments regarding why it would be beneficial, such as ease of availability and fewer authentication details needing to be remembered by users.

Their opinion was bolstered by comScore (2014), whose research showed that 65.5% of smartphone users download an average of 0 apps per month over a three-month period. Additionally, the research also found that 42% of all time spent on smartphone apps is spent on the user's single most used app, further reinforcing the potential for a central application within which users can access many functionalities.

All of these sources strongly suggest that the use of bots will continue to increase, with some even suggesting that bots will replace traditional applications. The evidence put forward to support these claims is fairly convincing, with tech giants such as Facebook backing up the statements. The research by comScore (2014) also appears to be reasonable evidence that bots beginning to replace traditional applications is likely. Therefore, a further increase in popularity of bots seems probable, and so tools to make operating bots more efficient will almost certainly prove useful, which serves as additional justification for this project.

Bots are being developed to provide increasingly useful functionality. However, this development comes at a cost. Bots are widely used for unethical purposes, such as DDoS attacks, email harvesting, and skewing political conversation (Howard, Kollanyi, and Woolley 2016). Howard, Kollayni, and Wooley detailed the study they conducted, where an analysis was performed on bot accounts' tweets around election day for the 2016 U.S presidential election. The study analysed 18.9m political tweets that were posted by bots, concluding that 55.1% of the tweets were pro-Trump while 19.1% of them were pro-Clinton.

The authors explained most of the study's methodology and findings in detail. However, a bot account was defined as any Twitter account that posted at least 50 times per day throughout the 9-day study, using at least one of the chosen election-related hashtags. This number of posts appears to have been chosen arbitrarily, as no solid reasoning was given for the choice, except the authors' claim that the likelihood that an account producing this number of tweets being a bot is "probable". Due to this decision, there may have been a significant number of accounts included in the analysis that were not bot accounts, or many bot accounts may not have been included. These factors reduce the study's credibility.

Nevertheless, it is likely that at least a large portion of these accounts were bots, highlighting the potential for bots to be used in an unethical manner. This suggests that developing a method of making bots more accessible needs to be carefully thought on before being implemented due to the ways that it could potentially be used dangerously.

5.4 Discord Bots

The literature on Discord bots is limited. However, a study of 300 Discord servers by Kiene and Hill (2020) found that the maximum number of bots present in any of the servers analysed was 20. A more intuitive and efficient way to configure a large number of bots such as this contributes to the motivation for this project. The sample size of n=300 in this study was small relative to the total number of Discord servers, but the research still provides a good overview of the number of bots that are typically present within a server.

The study by Kiene and Hill revealed that the mean number of bots in a Discord server was 3.31 (2.75 after re-weighting to compensate for over-sampling). This may appear to suggest that developing a tool to configure bots in a Discord server is unnecessary due to the low number of bots that are typically present. However, the research does not state how many of these bots were configurable; even if there was only one bot in a server, if it had 30 configurable parameters, then the system that this project intends to develop would be very beneficial. Additionally, this project is partially aimed towards administrators of servers that contain a wider selection of bots, each of which may have some configurable parameters, so the fact that some servers may contain up to 20 bots alone remains adequate motivation for development.

5.5 Existing Projects

Arcane (Arcane 2021) is a Discord bot that advertises "Leveling, xp, ranks, voice, role rewards, auto mod, reaction roles, custom commands, Youtube alerts", as well as a "simple dashboard for managing rewards, levelup notifications, and xp options" (Top.gg 2021a). The dashboard that Arcane provides is intuitive and allows configuration of the bot, similar to the interface that this project intends to create. However, the dashboard is only compatible with Arcane, while the system created in this project aims to be compatible with any general Discord bot.

Dyno (Dyno 2021) is a similar bot to Arcane, offering a "fully customizable bot for your server with a web dashboard, moderation, autoroles, automod, reaction roles, starboard, and more" (Top.gg 2021c). The dashboard that Dyno offers is intuitive and allows a massive number of settings to be configured for the bot. The interface that this project aims to create may take inspiration from the impressive configurability that Dyno's dashboard offers. However, similar to Arcane, the dashboard that Dyno offers only allows control of the Dyno bot – a problem that this project aims to overcome.

Discord Bot Dashboard (Yaman 2021) is another existing project that has some similarities to the system this project aims to develop. It allows bot developers to install a package and add some code to their bot that produces an online dashboard while the bot is running. The dashboard presents analytics for a bot to its developer. This is similar to the system this project aims to develop in the sense that it allows integration into individual bots and provides an online interface. However, Disco Bot Dashboard is aimed solely towards bot developers, to allow them to monitor the usage of their bot, whereas the system created during this project aims to be used by both bot developers and server administrators. Additionally, Discord Bot Dashboard does not allow configuration of the bot, whereas the system created during this project will.

Therefore, it appears there are no existing projects that possess the functionality this project aims to create, so there is strong justification for the project, and a potential for the system developed during this project to be beneficial to Discord users.

6 Approach

6.1 Project Management

Trying to come up with a definitive timeline for this project is difficult. This is partly since I have previously used only one of the four technologies that will be utilized. I'm not aware how long it will take me to first learn how to use these technologies so trying to set exact dates for the rest of the project's requirements seems futile. The requirements set out in the Aims and Objectives chapter are the only plans that I have, and even these are subject to change based on how the scope of the project shifts during its completion. Therefore, the approach I will take on this project will be similar to what Cho (2021) describes in their article 'What the Fog':

> When caught in a fog, it's a natural instinct for achievement-minded people to grope wildly around for something concrete to hold onto, or to plan toward. However, planning requires some level of known conditions and known objectives—and that's just something you may not have. In those situations, planning is truly an empty exercise. You're doing it because you feel the need to, but not because it's what will unlock you.

Cho's definition of fog is "disorientation" and an "inability to see clearly". Although this extract doesn't describe my situation exactly since I do have some known objectives, I can still relate to this feeling since I will be exploring several unknown technologies and using processes from many fields that I have no current knowledge of. Therefore, I will not create a concrete timeline for the project, but will just aim for the end goal.

6.2 Web Application

6.2.1 The Stack

The system will be built using the popular MERN stack: MongoDB, Express, React and Node.js. This means the online graphical interface that Discord server administrators will use to configure bots in their servers will be built using React, a powerful front-end Javascript library, commonly used for building single-page applications; this is the only of the technologies I have experience using. The power of React is demonstrated by its use in many large companies, including Facebook, Netflix, Reddit and Dropbox (AnyforSoft 2021). This evidence of the utility of React and the positive experiences I have had using it are the reasons I have chosen it for this project.

The backend of the application will be built using the Express framework; this backend will be an API that allows the frontend, the database, the bots and the Discord API to communicate. Express is known as the de facto standard web application framework to use in Node.js and is commonly used in conjuction with React (Serby 2012). Express and React synergize well because they are both Javascript based, meaning only one language is required to develop with both of them.

It is planned that the system's database will be built using MongoDB. This is a NoSQL database program that allows rapid development and quick prototyping due to being non-relational and thus inherently flexibile. MongoDB was chosen primarily because it is part of the MERN stack - it synergizes with the other technologies since it stores data in a JSON-like format, which works well with the rest of the Javascript-based stack. Additionally, I thought it would be beneficial to learn how to use a NoSQL database for my own personal development since I have never used one before. If necessary, a more robust database management system, such as PostreSQL or MySQL can be utilized, depending on how far the project is developed.

And of course, the Javascript runtime environment Node.js will be required to power the application since Express and React are both Node.js packages. Also, the framework that bot developers will use to integrate their bots with the system will be written in Node.js since the bots created during development of the system will be Javascript based. This completes the MERN stack.

6.2.2 Alternative Considerations

I considered using Python's Flask web framework for the frontend and backend of this project since I have a good amount of experience using it for these purposes. However, the single-page functionality that React offers provides a very smooth, and in my opinion, superior, user experience, so it was chosen for the frontend. I also want to develop my skills using React since it is widely used in the software industry so the skills acquired are likely to be beneficial in future.

There are other libraries and frameworks capable of creating single-page applications, such as Angular and Vue.js, but I have prior experience using React while I don't have experience using the others, and so React felt like the logical choice.

After deciding to use React for the frontend, it seemed natural to use a Javascript-based backend, and so I opted to use Express, the most popular of these. There are other alternative frameworks that could have been used, such as Java's Spring or Python's FastAPI, but I have no experience using these either and they are written in different languages to React, likely slowing development time; therefore, I saw no reason to use any of these over Express and so decided Express was the best choice.

6.3 Bots

The two choices considered for developing the bots used for testing the system were: Javascript with the discord.js library, and Python with the discord.py library; these were the choices considered since these are languages I have experience programming in, are my favourite to use, and appear to have the biggest Discord bot developer communities.

From my experience, both the discord.js and discord.py libraries offer simple and efficient communication with the Discord API. However, using discord.js is more in-line with the rest of the technology being used, since the stack is primarily Javascript-based. Additionally, I have preferred the style and usability of the Javascript version in my experience so far.

Therefore, the system will be developed using the Javascript-based package. However, the Python version may be utilized if limitations are found within discord.js that are not present in discord.py.

This does mean that the project intends to only allow integration with Javascript-based Discord bots. However, if the project proves to be sufficiently successful, a system that integrates with bots developed in more languages, such as Python, may be developed.

6.4 Hosting

During development, I will locally host the React frontend, Express backend, and any bots that are developed, as this will allow for more rapid development than pushing to a production build on an externally hosted server whenever a change is made. It will also reduce the cost of having to pay for hosting. MongoDB offers free hosting via MongoDB Atlas, which I will use.

If the project is developed far enough, the frontend, backend, and bots will be hosted on an external server in a production environment. The candidates for the hosting service include Cardiff University's OpenStack server, Heroku, Amazon EC2, Vultr, and Qovery. It would be preferable if Cardiff University's OpenStack server could be used for hosting, as this may grant me support from the University's tech support if required and is also free to use. If this choice proves not to be practical, another option can be used. All the other options were chosen because they are popular hosting choices, except for Qovery – this is an option primarily because it is free to use.

6.5 Evaluation

Evaluating the solution that is created will be done in several ways; one of these will be extensive unit testing, where all functions that are written in the system are tested. This will ensure that all components of the system function correctly when used independently.

There will also be end-to-end testing, where typical usage in production is simulated. This will ensure that the system operates correctly when used for its intended purposes. This testing will likely be performed by me; I will create a list of functions that the system should correctly perform, and test each of them. At the current time, I am unable to say exactly how the system will operate and so I am unable to give a list of these functions.

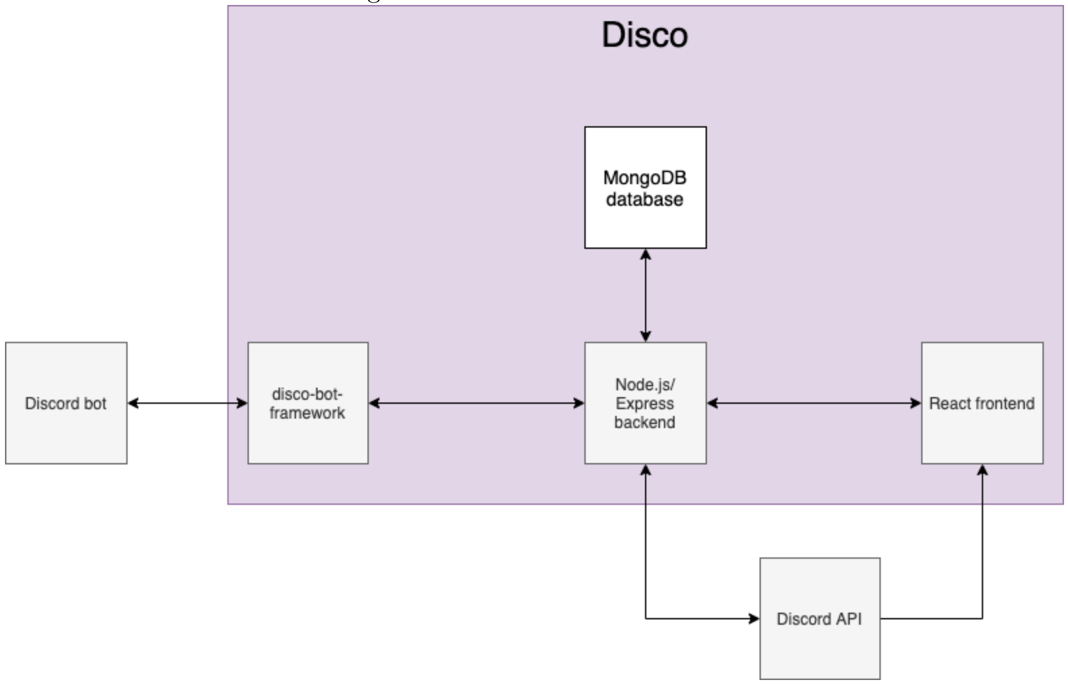
Finally, there will be usability testing, where participants will be recruited to test the system. The feedback from the user testing will be used to evaluate how usable and intuitive the system is to use, and whether it is more convenient than configuring bots in the traditional manner.

7 Design and Implementation

During development of the project, 'Disco' was used as a working title for the project, and so the system may be referred to by this name throughout this report.

The architecture of the completed system can be seen below, as well as how it interacts with a Discord bot and the Discord API.

Figure 7.1: Architecture of Disco



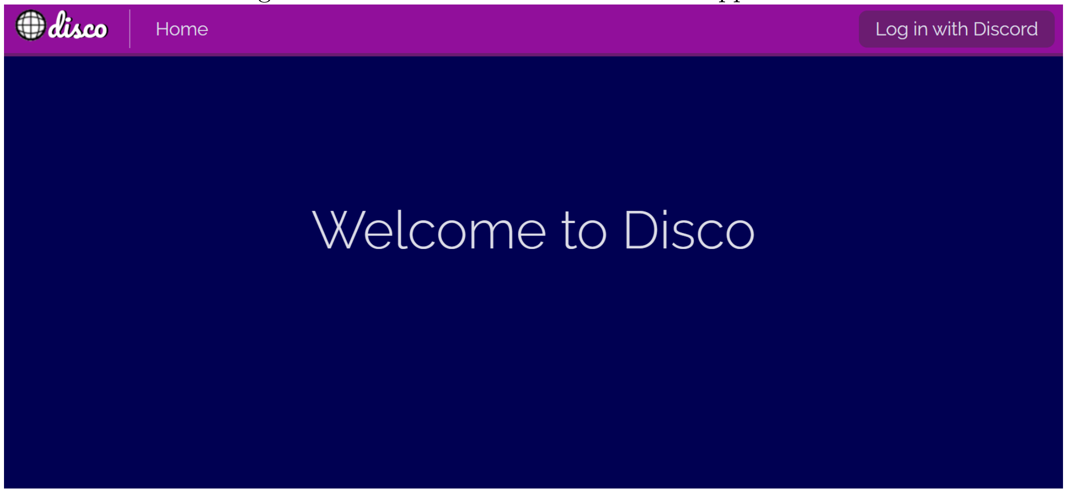
As can be seen in the figure above, Disco consists of four products, a Node.js/Express backend, a React frontend, a MongoDB database and disco-bot-framework, which is a Node.js package. The connecting arrows represent the directional exchange of information. So we can see the Node.js/Express backend connects all parts of Disco and allows them to communicate; it does this via an API. Additionally, we can see disco-bot-framework is what connects a Discord bot to the system.

The diagram also portrays the modularity of the system. Each of the products within Disco is an application in its own right and has potential to be used in other systems.

7.1 Frontend

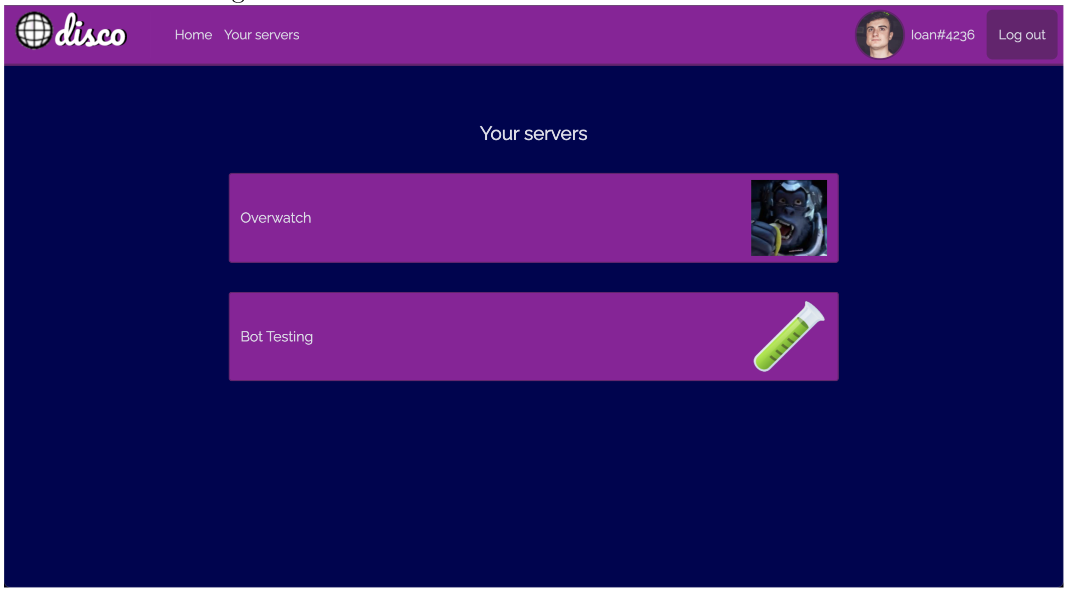
The frontend of the web application was implemented using the React library. As can be seen below, the home screen offers the option for the user to log in with their Discord account. Login via Discord was manually implemented using the OAuth2 authentication that the Discord API offers.

Figure 7.2: The home screen of the web application



Once logged in, the user is able to see the servers that they are an admin of by clicking on ‘Your servers’. This is done by making a GET request to the Express.js backend server, which, in turn, calls the Discord API to retrieve the list of servers that the user is in, and then filters to the ones that the user is an administrator of.

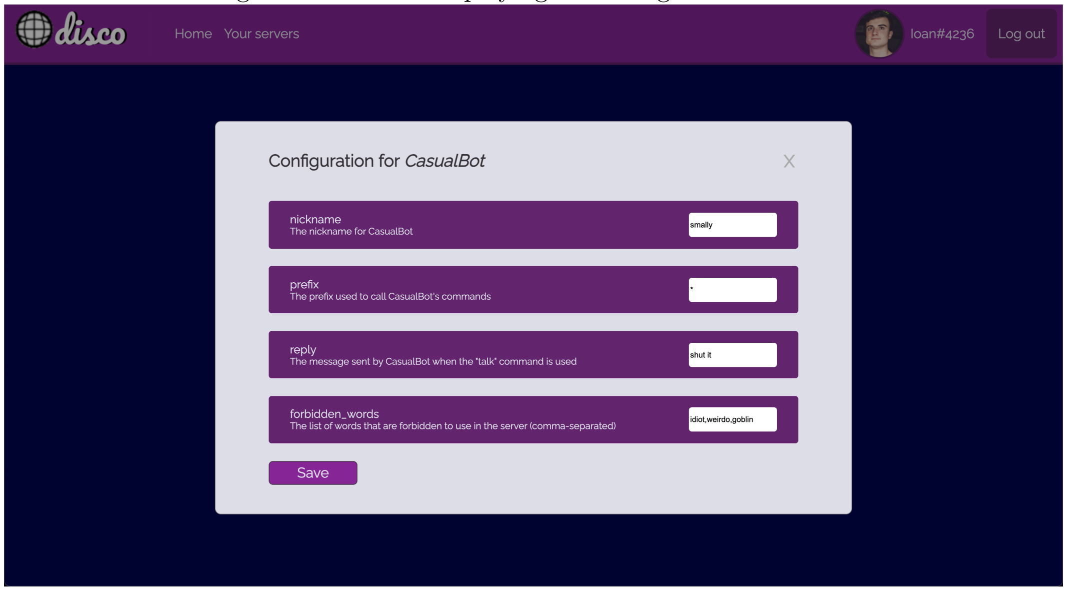
Figure 7.3: List of servers the user is an administrator of



After selecting a server, the user is shown the list of bots in the server that are registered in the system's database. This is done by making a GET request to the backend server, which, in turn, queries the database for the list of bots registered with the system that are a member of the selected server.

The list of registered bots is shown to the user. The user then chooses a bot and is shown a modal displaying the current configuration of the bot. This modal displays the name of each configurable parameter, its description and its current value, which can be edited.

Figure 7.4: Modal displaying the configuration of a bot



Configuration for *CasualBot*

nickname The nickname for CasualBot	smally
prefix The prefix used to call CasualBot's commands	*
reply The message sent by CasualBot when the "talk" command is used	shut it
forbidden_words The list of words that are forbidden to use in the server (comma-separated)	idiot,weirdo,goblin

After editing the values of the parameters as desired, the user clicks 'Save', and the changes will be reflected in the database.

7.1.1 Accessibility

Accessibility was an important factor to consider when developing the web interface. Therefore, the website is compatible with some of the main accessibility requirements laid out by Henry (2021). This means that all features of the website are fully usable with only a keyboard, to allow users who are not able to operate mouses to use the application. Additionally, the `alt` HTML attribute was set on all images, to allow users of screen readers to understand what images are portraying. Furthermore, an effort to use semantic HTML was made, to allow users of screen readers to more easily interpret the structure and content of the application.

7.2 Backend

The back-end Express server is the backbone of the system. It connects the bots to the database and connects the frontend to the database, therefore connecting the bots to the frontend, allowing for their configuration. The main interactions with the Discord API are also performed by the backend, hence connecting the rest of the system to it.

7.2.1 Requests

The backend server utilizes a REST API. I chose this design since it is a common standard for online APIs, it is intuitive to use and this project is small enough that it was the best option. I considered using GraphQL for the structure of the API, but it seemed overly complex for the small size of this project. The way that the server conforms to the definition of a REST API is shown below.

Note: A `config` resource is the configuration of a particular bot within a particular Discord server

- A GET request to `/api/db/config` returns either:
  - All the `config` resources for a particular bot
  - All the `config` resources for a particular server

  Note: Which one is returned depends on the query parameter in the URL of the request. If the URL contains `botID` as a query parameter, all the `config` resources for that bot are returned. Whereas, if the URL contains `guildID` as a query parameter, all the `config` resources for a particular server are returned. If the URL contains both the `botID` and `guildID` query parameters, the API ignores the `botID` query.
- A GET request to `/api/db/config/:configID` returns a single config resource based on the `configID` parameter

  Note: `configID` is a unique identifier for a configuration, generated based off the ID of the bot that the configuration is for and the ID of the server that the configuration is for
- A POST request to `/api/db/config/:configID` creates a `config` resource with an ID of configID, based on the data in the body of the request. These requests are made by `disco-bot-framework` when a bot implementing the package is invited to a server

- A PUT request to `/api/db/config/:configID` replaces the `config` resource with an ID of configID with the data in the body of the request. This is done when a server administrator alters the configuration of a bot via the front-end interface

Originally, the main resource of the API was a `bot`, and so requests were make to `/bot/...`. This, however, meant that when, for example, retrieving the configuration for a particular bot, problems arose if a bot was a member of multiple servers since the system could not differentiate between the configuration of a bot in two different servers. The solution to this was the focus the API around `config` resources, which are unique for a particular bot in a particular server.

I used the API development platform Postman to test the endpoints in the backend throughout the project, as this allowed me to make direct requests to the backend without worrying about the implementation in the frontend. This made it easier to track down bugs and problems, since if I was using Postman, I knew the error must be in the backend since the frontend was not being used.

7.2.2 Discord API

After doing some research on the Discord API, I decided the best way for users to log into the web application is to use their Discord account and the OAuth2 protocol. OAuth2 allows third-party applications to obtain limited access to a user's account (Anicas 2014). I decided to use this protocol because it is known to be secure and is the standard for implementing authorization online. Additionally, it reduced development time since a custom authorization solution did not need to be manually implemented.

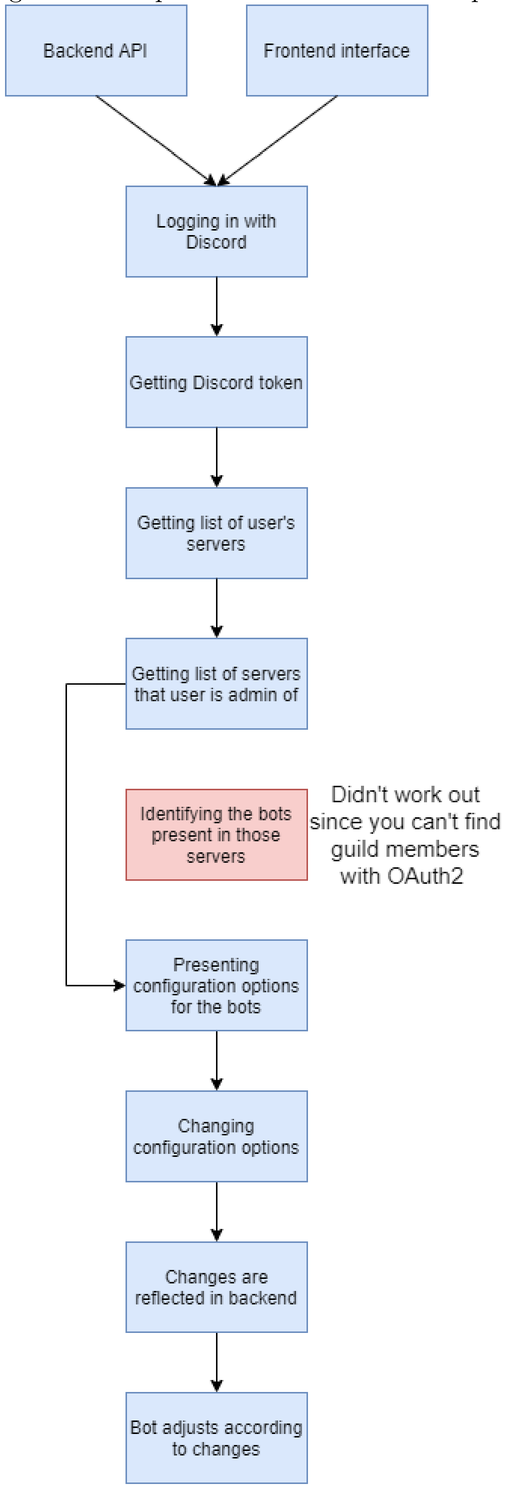
There are several steps to this authorization flow. Firstly, a user clicks the 'Log in with Discord' button in the frontend; this allows the user the authorize the Disco application, which is registered as a Discord application, to limited access to the user's Discord account. After the user authorizes the application, a `code` is added as a query parameter to the URL that the user is redirected to (the redirect URL takes them to the home page of the frontend). After this, the frontend makes a POST request to the the backend at the endpoint `/api/discord/token`; the request's body contains the `code` that was returned in the previous step. Next, the backend makes a POST request to the Discord API, with the body of the request containing the client ID of the Disco application, the application's secret, and the `code`. The Discord API responds by sending an access token that allows the application to make requests to the Discord API on the user's behalf; this access token is then returned to the frontend for use.

Additionally, the logout feature of the website utilizes the backend. When a user clicks ‘Log out’, a POST request is made to the backend’s endpoint `/api/discord/token/revoke`, with the body containing the current access token. The backend then makes a post request to the Discord API, with the body containing the client ID of the Disco application, the application’s secret, and the access token. The access token is then revoked, meaning it can no longer be used by the Disco application to make requests on the user’s behalf.

7.2.3 Planning

I came up with the following rough list of tasks that would need to be completed during development of the backend and frontend. Most of the tasks relate to the backend, so this feels like the most appropriate section to discuss it. Note that the list was designed to be used by myself and so contains informal language, but the meaning is still there.

Figure 7.5: Sequential list of tasks to complete


Backend API
Frontend interface
Logging in with Discord
Getting Discord token
Getting list of user's servers
Getting list of servers that user is admin of
Identifying the bots present in those servers
Didn't work out since you can't find guild members with OAuth2
Presenting configuration options for the bots
Changing configuration options
Changes are reflected in backend
Bot adjusts according to changes

As can be seen in the diagram, my initial plan was not successful. I was under the impression that after authorizing the user with OAuth2, I could use the backend to send a request to the Discord API to retrieve the list of users in the server that the user had selected. The official documentation is not particularly detailed on this matter; however, after some research, I came to the conclusion that this endpoint is only accessible by bot applications (MinnDevelopment 2020), so I had to find another way to present the list of bots that are present in their server. In the end, I decided it was more sensible anyway to only show bots that were registered with the system, and so the backend now queries the database for the list of bots instead of the Discord API. Note that this fix is not noted in the figure above.

Other than this setback, the list of tasks was completed successfully, and although fairly rudimentary, provided a solid direction and foundation for the application.

7.3 Database

The database is implemented using MongoDB, which is a NoSQL, document-oriented database program. The JSON-like documents that MongoDB uses integrate well with the rest of the system since the system is Javascript based. The database is hosted via MongoDB Atlas, the official MongoDB cloud hosting service; it provides stable, fast and free hosting.

To allow communication between the back-end server and the database, I initially utilized the official MongoDB driver for Node.js; this was capable of achieving what was required. However, there is an alternative Node.js package called Mongoose. Mongoose uses the MongoDB driver under the hood, but provides a higher-level API than the official MongoDB driver and, as a result, is simpler to use. This was an important factor when developing under the time constraints of the project, and so I chose to use Mongoose instead of the official MongoDB package to reduce development time.

Another reason for choosing the Mongoose package was due to the schemas that it can define since by default, MongoDB does not enforce schemas. These schemas allow a more structured dataset to be defined, which can help maintain reliability of the database.

To register a bot in the system's database, the bot simply needs to be invited to a Discord server, the bot framework then automatically makes a POST request to `/api/db/config/:configID` in the backend, which registers the bot in the database using the default configuration defined within the bot.

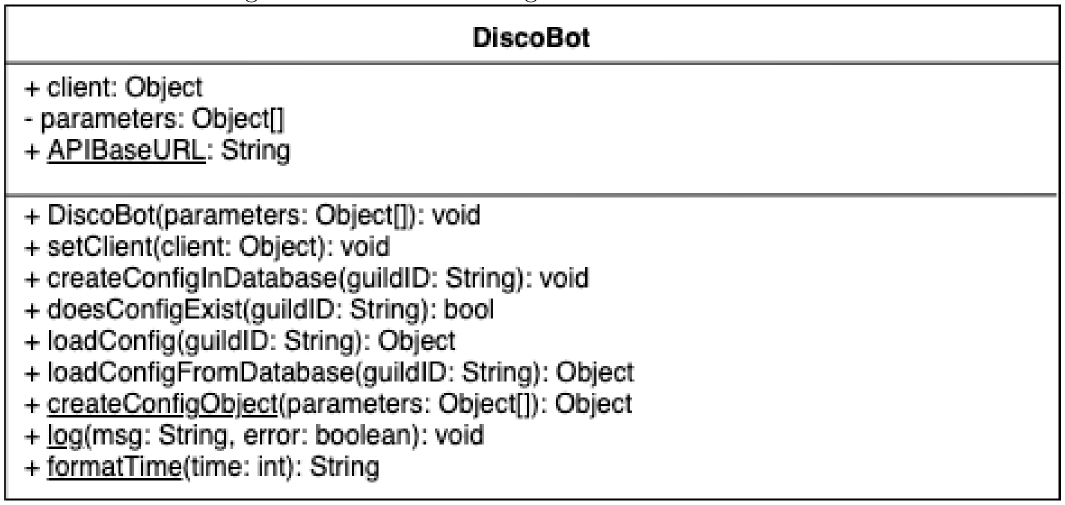
The system contains only one database, `discoDB`, which contains only one collection, `configs`. The schema for a single `config` is shown below.

```
{
  id: String,
  botID: String,
  guildID: String,
  botName: String,
  config: [
    {
      name: String,
      description: String,
      value: String
    }
  ]
}
```

7.4 Bot Framework

The bot framework is relatively simple. It consists of an `npm` package called `disco-bot-framework`, which has only one external dependency. This package contains only one Javascript file, which contains a single class `DiscoBot` with three properties and eight methods. The UML class diagram for the DiscoBot class can be seen below.

Figure 7.6: UML class diagram for the DiscoBot class


DiscoBot
+ client: Object
- parameters: Object[]
+ APIBaseURL: String
+ DiscoBot(parameters: Object[]): void
+ setClient(client: Object): void
+ createConfigInDatabase(guildID: String): void
+ doesConfigExist(guildID: String): bool
+ loadConfig(guildID: String): Object
+ loadConfigFromDatabase(guildID: String): Object
+ createConfigObject(parameters: Object[]): Object
+ log(msg: String, error: boolean): void
+ formatTime(time: int): String

7.4.1 Previous implementations

Originally, the bot framework was designed such that each bot that wanted to use the system had to directly connect to the system's database. This made integrating the functionality of Disco into a bot very complicated, as much configuration was required to incorporate the database functionality. This meant that many lines of code were required to integrate Disco into a bot, leading to a poor and potentially frustrating experience for the bot developer.

As an alternative to this, I made all requests to the database go through the backend server. This functionality was incorporated into the `disco-bot-framework` package which hides the complexity from the bot developer.

Another problem I encountered after this was the `disco-bot-framework` package originally consisted of a Javascript file containing only two functions, `registerBot` and `loadConfig`. A developer trying to incorporate the framework into their bot had to explicitly call these functions somewhere in the source code for their bot when some action happened. Additionally, the developer had to pass several arguments to these functions, further complicating the process of integrating the system into a bot; the `registerBot` function required three parameters: `clientUser`, `guildID` and `parameters`. The `loadConfig` function required the `botID` to be passed each time it was called.

To fix this, I refactored the functions into the `DiscoBot` class, which stores several proprties that prevent parameters needing to be passed each time a method is called. The class also contains several extra methods that help functionality. To overcome the problem of having to explicitly calling multiple functions, the framework is now programmed to automatically register the bot in the database when it is invited to a server; this reduces work required by the bot developer. The `loadConfig` function is still required to be called explicitly, but this is one simple line of code and can, for example, be triggered each time a message is sent to a channel in the server.

7.4.2 Properties and methods

The following is a listing and description of the properties and methods of the `DiscoBot` class.

Properties

client The `client` property is an instance of the `Discord.Client` class from the `discord.js` package. It is the instance of the actual bot client.

parameters The `parameters` property is an array containing the configurable parameters of the bot. Each element in the array represents a parameter and is an object with a `name`, `description` and `value - value` represents the default value of the parameter when the bot is registered in the database. For example, for the bot that was used while developing Disco, `parameters` is defined as:
Note: The username of the bot is ‘CasualBot’.

```
const parameters = [
  {
    name: ‘nickname’,
    description: ‘The nickname for CasualBot’,
    value: ‘CasualBot’
  },
  {
    name: ‘prefix’,
    description: ‘The prefix used to call CasualBot\’s commands’,
    value: ‘!’
  },
  {
    name: ‘reply’,
    description: ‘The message sent by CasualBot when the "talk" command is used’,
    value: ‘Hello!’
  },
  {
    name: ‘forbidden_words’,
    description: ‘The list of words that are forbidden to use in the server (comma-
        separated)’,
    value: ‘idiot,weirdo,goblin’
  }
];
```

APIBaseURL The `APIBaseURL` static property is the URL of the back-end server's API. I am responsible for keeping this property in sync with the actual URL of the backend.

Methods

DiscoBot This is the constructor method of the DiscoBot class. It has one parameter, `parameters`.

setClient This method sets the `client` property of the `DiscoBot` class from the `client` parameter it is passed. Additionally, this method registers a listener to the `guildCreate` event of `client`, which creates a new configuration in the database whenever the bot is added to a new guild.

createConfigInDatabase This method adds a new `config` entry to the database for the bot for a particular guild using the `guildID` parameter it is passed.

doesConfigExist This method takes one parameter, `guildID`, and returns a boolean representing whether a configuration already exists for the bot in the guild with ID `guildID` by making a request to the backend to check the database.

loadConfig The `loadConfig` method returns an object representing the configuration of the bot for a particular guild from the database based on the `guildID` that it is passed. This object can then be used directly by the bot.

loadConfigFromDatabase This method attempts to load the configuration of the bot for a particular guild from the database using the `guildID` parameter it is passed.

createConfigObject This static method takes a 2-dimensional array, `parameters`, representing a configuration. Each element of `parameters` is an array of size 2, where the first element is the name of a parameter in the configuration and the second element is the value of that parameter. The method uses the `parameters` parameter to return an object consisting of key-value pairs, where each key is the name of a parameter and each value is the value of that parameter.

log This static method takes two parameters: a string called `message` and a boolean called `error`. The method is used to print `message` in a timestamped format to either `stdout` or `stderr`, depending on if `error` is false or true. This is intended to aid the bot developer in debugging issues with their bot while it is using Disco.

formatTime This static method takes a number, `time`, converts it to a string and returns it padded to the left with zeroes such that it is of length two. This is intended to convert numbers to readable time values (representing either hours, minutes or seconds) for use by the `log` method. For example, it will convert the numbers `1, 5, 9, 15, 43` to the strings `"01", "05", "09", "15", "43"` respectively.

7.4.3 Implementing the functionality

Assume the developer already has a functioning, regular Discord bot. In the source code for the bot that the developer wishes to adapt to work with disco, the developer must:

1. Download install the `npm` package `disco-bot-framework`
2. Import the package into their bot using `require('disco-bot-framework')`
3. Define an array of parameter objects
4. Create an instance of the `DiscoBot` class, passing the array of parameters
5. When the instance of the `Discord.Client` class from the`discord.js` package (that they already have in their source code) emits the `ready` event, set the `client` property of the `DiscoBot` instance to the instance of `Discord.Client`

In practice, a conversion from using a regular bot to a bot using Disco looks like the following; this is a minimal example of a Discord bot using `discord.js`.
Note: in this example, the developer is storing the configuration in a file `config.js`, a fairly standard way to store the configuration of a Javascript program.

Listing 7.1: Bot implementation without disco-bot-framework

```
const Discord = require('discord.js');
const config = require('./config');
const client = new Discord.Client();

// Execute a function once the bot is ready
client.once('ready', () => {
  console.log('Client is ready');
});

// Execute a function when the bot detects a message has been sent to a server it is in
client.on('message', async message => {
  // Do not proceed if a bot sent the message or if the message doesn't begin with the defined prefix
  if (message.author.bot || !message.content.startsWith(config.prefix)) return;

  // Remove the prefix from the message; the remaining text is the command
  const command = message.content.slice(config.prefix.length);

  if (command === 'talk') {
    // Send the defined message to the channel that the command was send from
    await message.channel.send(config.message);
  }
});

// Log the bot into Discord, using the token defined in the config file
client.login(config.TOKEN);
```

Listing 7.2: Bot implementation with disco-bot-framework

```
 1 const Discord = require('discord.js');
 2 const config = require('./config');
 3 const DiscoBot = require('disco-bot-framework');
 4
 5 const client = new Discord.Client();
 6 const discoBot = new DiscoBot(config.parameters);
 7
 8 // Execute a function once the bot is ready
 9 client.once('ready', () => {
10   discoBot.client = client;
11 });
12
13 // Execute a function when the bot detects a message has been sent to a server it is
      in
14 client.on('message', async message => {
15   // Do not proceed if a bot sent the message
16   if (message.author.bot) return;
17
18   // Load the config from the database for the guild the message was sent to
19   const botConfig = await discoBot.loadConfig(message.guild.id);
20
21   // Do not proceed if the message doesn't begin with the defined prefix
22   if (!message.content.startsWith(botConfig.prefix)) return;
23
24   // Remove the prefix from the message; the remaining text is the command
25   const command = message.content.slice(botConfig.prefix.length);
26
27   if (command === 'talk') {
28     // Send the defined message to the channel that the command was sent from
29     await message.channel.send(botConfig.message);
30   }
31 });
32
33 // Log the bot into Discord, using the token defined in the config file
34 client.login(config.TOKEN);
```

As can be seen from this example, only six extra lines are code are required to implement `disco-bot-framework`:

1. Requiring the `disco-bot-framework` package on line 3
2. Creating a variable to store the instance of the `DiscoBot` class on line 6
3. Setting the `client` property of the `DiscoBot` instance on line 10
4. Loading the configuration of the bot from the database on line 19
5. Changing `config.prefix` to `botConfig.prefix` on line 22
6. Changing `config.message` to `botConfig.message` on line 29

After implementing this, administrators of servers the bot is invited to can use the web interface to configure the bot.

Of course, this is a minimal example, and a real bot would likely have far more configuration options so a few more changes would need to be made to implement `disco-bot-framework`. But this still demonstrates the idea of how implementation is done.

After these necessary simple steps, it is completely up to the bot developer to decide on how to use `disco-bot-framework` within their bot.

7.5 Deployment

To allow users to participate in testing the system, I needed to make the system publicly available. This meant hosting the frontend and backend publicly, as well as using a virtual machine to run the bot.

I decided to use DigitalOcean for all of these requirements, even though it was not listed in the candidates for hosting services in the Approach chapter. The main attraction of this service was that they offer $100 in free credit to students via the Github Student Developer Pack. On first inspection, I also found it very intuitive to use and so stuck with it.

I hosted the frontend at one public URL and the backend at a separate public URL. This cost slightly more than hosting them both wihtin the same application, which would only require a single URL, but I found that it helped to reduce coupling and kept the application as modular as possible.

I also acquired an Ubuntu virtual machine via DigitalOcean to run the bot. This was required because although it would have been possible to run the bot from my own machine, it would have meant my machine would need to be running 24/7, which would not be ideal.

7.6 Security

While developing a web application that is designed to be usable by the general public, security is of paramount importance. Users of the system will rely on the system to reliably store their data without corruption or unauthorized modification. There were several ways security measures were taken into account while developing the system.

7.6.1 OAuth2

OAuth2 was used for logging in to the web application via Discord. OAuth2 is the standard, most popular protocol for authorisation. It allows third-party applications to gain limited access to an HTTP service on behalf of a user (Hardt 2012). An important factor that had to be incorporated into this authorization flow was the use of a `state` parameter.

To log into the application using their Discord account, the user clicks ‘Log in with Discord’, which sends them to the following URL (separated across several lines for clarity):

```
https://discord.com/api/oauth2/authorize?
client_id=862356663351377952
&redirect_uri=https%3A%2F%2Fdisco-react-qavzn.ondigitalocean.app
&response_type=code
&scope=guilds%20identify
&prompt=none
&state={state}
```

This is the format of a URL that is required to use OAuth2 with Discord. There are a few query parameters here that need some explaining:

1. `client_id` is the ID of the Disco application, which is registered as a Discord application

2. `redirect_uri` is the URL that the user is redirected to after logging in with their Discord account

3. `response_type` is set to code, meaning a code is returned after the user authorizes. This is used to exchange for a token in the authorization flow

4. `scope` is used to define the scope of interactions the application can make with the Discord API on behalf of the user. It is important that only the necessary scopes are asked for to ensure only the required permissions are granted

5. `prompt` is set to none, meaning if a user has authorized with the application, they do not need to reauthorize when visiting the web application again unless their access token expires

6. `state` is the most important parameter for security. The value of the `state` query parameter is unique for each user session. After authorization, when the user is redirected, if the `state` query parameter matches the original `state`, then the user is properly authenticated. This helps prevent Cross Site Request Forgery and Clickjacking vulnerabilities. If the returned `state` does not match the original `state`, then it’s possible that someone interrupted the authorization request, and so the request should be denied (Discord 2021c)

7.6.2 React

There are other attacks that need to be considered when developing a web application. For example, injection attacks are a popular form of attack, where malicious code is injected into user input. React prevents injection attacks by default, sanitizing all user input by escaping embedded values. This helps prevent a common type of injection attack known as cross-site-scripting. (React.js 2021c).

There have been some notable cross-site-scripting vulnerabilities within React, but these have since been patched, resulting in React being a generally secure library (Mueller 2017).

7.6.3 Backend

Since the backend of the website is hosted publicly, it is theoretically possible for anyone to send a request to the API. Without any security measures, this means that anybody could update the configuration of a bot in a server if they knew the ID of the server, which is easily retrievable via the Discord API.

To combat this, when a PUT request is made to `/api/bots/:configID`, the backend checks the `Authorization` header of the request. For the request to be successful, the `Authorization` header must contain an access token, retrieved from the Discord API, of a user who is an administrator of the server that one of the configurations is attempted to be updated for. If the token in the `Authorization` header is missing or invalid, the backend sends a response of `400 Bad Request`, with the message `{ "error": "Token missing or invalid" }`. This ensures that only administrators of a server can update the configuration of bots in the server. Additionally, the backend implements a Cross-Origin Resource Sharing (CORS) policy. This limits cross-origin Ajax requests to only specified domains, helping increase the security of the application (Zakas 2010).

8 Evaluation

8.1 Addressing the requirements

I believe the project was a success in most areas. Consider the requirements laid out in the Aims and Objectives chapter:

1. The system must allow server administrators to intuitively configure supported bots using the web interface
2. The system must allow developers to integrate the functionality into their bots in an uncomplicated manner
3. The system must be secure, to prevent unauthorized alterations to bots in the database

The first two requirements were fully met. Regarding the first, it is simple for server administrators to use the system; the process is described in detail in the section 7.1 Frontend. To summarise, the steps are as follows:

1. Invite the bot to a server they are an administrator of
2. Navigate to the URL of the frontend (URL is not explicitly given here because it may change)
3. Select 'Log in with Discord' and proceed to log in
4. Select 'Your servers'
5. Select a server
6. Select a bot in that server
7. Adjust the configuration parameters of the bot
8. Select 'Save'

Therefore, there are only eight simple steps that need to be followed to configure a bot that supports the system. For the vast majority of users, especially those who are an administrator of a Discord server, this is very intuitive. However, it may not be as intuitive as possible for users with accessibility requirements since I was not able to implement as many accessibility features into the frontend of the application as I would have liked to. (React.js 2021a) provides a detailed article on how to implement necessary accessibility features into React applications. However, there are a lot of features to be considered for full accessibility, and due to the limited time frame of the project, I was not able to implement all of these. Some of the accessibility features I was not able to implement as fully as I would have liked are:

- Fully complete aria-* HTML attributes - these are attributes for HTML elements specifically designed to aid users with accessibility requirements to identify features of a web page designated for user interaction by, for example, describing a type of widget as a "menu", "progressbar", etc. (Cooper 2020). This can help users of screenreaders to more easily determine what a section of a webpage's function is.

- Perfectly semantic HTML - semantic HTML means using appropriate HTML elements to accurately describe the structure of a web page. For example, this means appropriately using the `<h1>` to represent top-level headings in a page, as opposed to, say, simply wrapping all different sections inside `<div>` elements. This, again, aids users of screenreaders to determine the structure of a web page. I did make an effort to fully use semantic HTML, but some sections of the web application do not abide by this; I was limited by the time frame of the project.

- Perfect use of titles - the `<title>` HTML element is used to set the title of a webpage. Accurate titles help users of screenreaders to more easily determine what a specific page is for. I did not achieve this, and so the title is simply 'Disco' for all pages of the website.

- Best colour choices - I attempted to design the interface using a simple colour pallete; this aids users with visual impairments in operating the website. However, on reflection, the two main colours used throughout the website are dark blue and purple. These colours do not contrast particularly well, potentially making it hard for some users to differentiate between different sections of a webpage.

Regarding the second requirement, it is simple for developers to integrate the functionality system into a bot that they have created. As described in subsection 7.4.3 Implementing the functionality, a developer need only follow five simple steps to have their bot working with the system:

1. Requiring the `disco-bot-framework` package
2. Creating a variable to store the instance of the `DiscoBot` class
3. Setting the `client` property of the `DiscoBot` instance once their instance of the `Discord.Client` class is ready
4. Loading the configuration of the bot from the database
5. Using `discoBot.*` for configurable parameters in the source code. For example, `discoBot.prefix`

The third requirements was met in some aspects, but not as fully as I would have liked. This is particularly relevant in the backend of the web application. The system was hosted publicly for testing, however, the link was only shared in the Discord server for my MSc Computing course. This is because if the link was readily available to the general public, any person could attempt to make requests to the API. Since the API has access to the database, the integrity and security of the data rely on it. Although a CORS policy was enforced, I am not knowledgeable enough on the subject to be sure that would would stop unwanted requests to the API.

As discussed in subsection 7.6.3 Backend, I implemented a security feature for PUT requests to the `/api/db/:configID` endpoint that means only administrators of a server can change the configuration for bots in their servers. However, similar security was not implemented for GET or POST requests to this endpoint, meaning anyone is able to potentially retrieve or create the configuration for any given bot in any given server as long as they know the ID of the bot and the ID of the client, which are easily accessible via the Discord API. This is not desirable since retrieving a configuration may, for example, allow an attacker to find the list of words that are forbidden in a server, allowing them to find abusive language that is not covered in this configuration. I did begin implementing a system where admins of servers could generate 'Authcodes' using the front-end interface, which would be used to authorize GET and POST requests sent to `/api/db/:config`. However, this was clunky and time-consuming, so I decided it was not feasible to implement it into the project given the time constraints.

Additionally, there is no rate limiting implemented in the backend. This means it is susceptible to denial-of-service attacks, where an attacker floods the API with bogus requests, leaving legitimate users unable to operate the system.

If the system is ever to be made intended for use by the general public, these security issues would need to be addressed before release.

Another suggestion for improving the project is that it may have been beneficial to conduct more research before beginning it. Although I believe the project is fully justified, I'm sure there are some people who are skeptical of this, partly due to the project covering such a niche topic. Therefore, if I had completed a study using Discord administrators or bot developers to determine whether they thought it was likely they would find this system benifical, the justification of the project may have been reinforced.

8.2 Testing

In the Evaluation section of the Approach chapter, I stated I would conduct extensive unit testing on the application. At this time, I was not aware of the complexities of testing React applications and APIs. I had conducted some unit testing on simple programs before, but my experience of testing systems similar to this one was nil. After researching methods of testing React application, it appeared that the aspect that would take the most time would be learning how to use a testing framework for React. Having never used one before, it seemed there was a large amount to learn before effective tests could be written.

I weighed up the pros and cons of learning one of these frameworks, and decided that due to the fairly limited interface of my frontend, learning an entire framework to test it would not be an efficient use of time. The frontend has a single primary workflow, which is logging in, navigating to the 'Your servers' page, selecting a server, selecting a bot, changing values within the configuration of a bot, and saving these changes. Instead of writing automated tests for all of my React components, I decided a more efficient use of time would be to manually perform end-to-end testing by performing the actions described above. Since this is the primary purpose of the application, if this workflow works correctly, then the frontend as a whole works correctly.

After manually conducting this end-to-end testing, I found that the application worked as desired. The application does not allow much user input and so there is little chance of erroneous data being entered by a user; the only data that a user is able to input is when they are changing the configuration of a bot. Extensive testing of inputting edge cases into the configuration values was not conducted, and so this is something that needs to be completed if the application is ever intended to be used by the general public.

Additionally, the backend did not receive as much testing as I would have liked. Ideally, tests should have been written to test all endpoints of the API to ensure they function correctly, but similar to what is described above, testing APIs that interact with databases is not trivial, and so the time limit imposed by the deadline of the project meant that this could not be feasibly included.

8.2.1 User testing

After receiving ethical approval, I distributed a questionnaire that asked users of the MSc Computing Discord server to test the system I developed; I did not receive any responses from this. I belive there may be a failing here on my part for several reasons. Firstly, there were external circumstances going on in my life that resulted in me distributing the questionnaire quite close to the deadline of the project; perhaps if more time was given for users to participate, I would have received some responses.

Secondly, the questionnaire I distributed required participants to complete a series of seventeen instructions before being able to answer the questionnaire, such as inviting a bot to a server they are an administrator of and then using the web interface to configure the bot. When faced with this instructions, I believe it is quite likely that some potential participants did not want to continue with the process due to is possibly taking too much of their time. I'm not completely sure of a solution to this since I made the instructions as simple and clear as was possible for properly testing the system.

Finally, due to the some of the security issues I mentioned in section 8.1, I was not confident in distributing the questionnaire publicly since the instructions include directly using the system. If I had implemented all necessary security into the system, then I may have felt confident posting the questionnaire to a wider audience and potentially would have received responses.

8.3 Preparations for the future

One of the main areas of future work identified is to do research on Discord's Slash Commands to see if this system would work with them. During the life of this project, Slash Commands gained a huge amount of popularity and now appear to be the standard method for Discord bots to receive commands, since the discord.js guide has been updated to include them as the default method of using commands in Discord bots (Discord.js 2021). They appear to be a significant upgrade over traditional commands due to the reliability they provide, and so if this system is to be developed any further, work will be done to try to make Slash Commands compatible with the system.

Something interesting is that this framework could easily be extended to work with Discord bots written in other program languages. For example, the only aspect of the project that would need to be adapted to allow Discord bots written in Python to use the system would be to create a version of `disco-bot-framework` written in Python. I believe this would be fairly simple, especially since Python is the programming language I have the most experience with, and given more time, I would have done this during the project. However, providing the proof that the application is viable was the main goal of the project, and so developing the system to work with bots written in Python is left for further work.

Due to the way I built the React frontend, many components were created to make the website function. The nature of these components makes them inherently independent and modular (React.js 2021b), and I believe some of them could be reused for other projects. Some examples of reusable components include the navigation bar and the items it contains.

Additionally, I suspect the system as a whole could be adapted to work with bots for applications other than Discord. Several other popular platforms incorporate bots, such as Teams (surbhigupta et al. 2021) and Facebook Messenger (Facebook 2021). Although some back-end code would need to be rewritten to allow the system to communicate with an API other than the Discord API, the same architecture shown at the beginning of the Design and Implementation chapter would likely be applicable to the system, since it is very general. However, this is far outside the scope of this project, and so is left as further work.

Another area for future work is improving the user experience of the frontend on mobile devices. The frontend was developed on a laptop and so is currently optimized for this type of device. Due to the prevalence of mobile devices, being able to use one to configure bots would likely be beneficial, and so this has been identified as an area for future development.

This could either be achieved through making the frontend more responsive to devices with varying screen sizes or by utilizing React Native. Although I have no experience using React Native, I know it is a tool that can be used for development of native apps for mobile devices by utilizing the React library.

Therefore, optimizing the front-end interface for use on mobile devices, either through a native application or a web application, should not be particularly challenging.

A problem that has been identified with the system is that frontend URLs other than the root URL `/` do not work correctly. The application utilizes `react-router-dom`, which is a library that loads React components based on the URL in the address bar. This functionality works as intended in a development environment, but it does not behave correctly in production. The problem is as follows: if a user is on the website and selects 'Your servers', the URL changes to `/servers` and they are shown the correct page. However, if a user tries to directly navigate to `/servers` without clicking 'Your servers', or if the user refreshes the page while on `/servers`, they are shown a `404 Not Found` error. After researching this problem, it seems that since the React application is being served as a static site, when the user navigates directly to `/servers` (or refreshes) without clicking 'Your servers', the server looks for a file called `servers`. However, since there is no actual file called `servers`, it does not load and instead shows a `404 Not Found` error (Pyltsyn 2020).

I have read about some solutions to this problem, but since it is not a critical issue (since a user never actually needs to refresh while on the website, and it is simple to begin using the website from the home page), I decided it was not worth using what limited time I had left to fix it.

Finally, a small improvement to the system would be that the configuration of a bot could be updated as soon as a change in its configuration is detected in the database, but the only ideas I've had to implement this would require the bot or bot framework to implement some sort of API that could be notified when a change in the database in detected. This seemed like it would require too much development time to implement during the project, and so is left as further work.

9 Conclusions

At the beginning of this report, I identified the problems that exist regarding configurable Discord bots; these being that using text commands to configure a bot is inconvenient, and developing a bot to be configurable via an online interface is very complex.

As explained in the Evaluation chapter, the requirements of the system were mostly met, the only exception was the security of the system, which was not implemented as well as I would have liked.

Therefore, in this dissertation, I have identified a problem that needed to be solved, justified why it was a problem, and addressed it, albeit not perfectly.

The achievements of this project are:

- Created a framework that allows bot developers to integrate the system into their bots
- Created an online interface that allows server administrators to configure supported bots in their servers
- Created a database to store the configuration of bots that are used in the system
- Created a REST API that:
  - Connects these various parts of the system together
  - Connects the system to the Discord API
- Therefore, created a solution to some of the complexities of configurable Discord bots
- All of these products are modular and have potential to be used in other applications

The deficiencies of the project are:

- The system is not completely secure
- The system has not been fully tested
- Some may think the project lacks motivation due to lack of research

The deliverables produced as a result of this project are:

- An npm package that allows bot developers to integrate the system into their bot
- A React application that allows server administrators to configure supported bots in their servers
- An Express server that connects the whole system and allows it to communicate with the Discord API
- A MongoDB database to store the configuration of bots for each server they are a member of

The main areas idenfitied as further work includes:

- Ensuring the system functions correctly when used with Slash Commands
- Allowing the system to be integrated into Discord bots developed in languages other than Javascript
- Researching whether the system could be adapted for use with bots in other platforms
- Optimizing the front-end interface for use on mobile devices
- Fixing the URLs on the frontend to work correctly when directly navigated to or when the page is refreshed

This project contributes to the body of existing knowledge surrounding Discord, bots, and Discord bots by proving that it is possible to create a system for configuring a general Discord bot via on online interface, and that it is not necessary for bot developers who wish for their bots to be configurable to create their own server, database or frontend since the work has been done for them.

10 Reflections

I learned a great deal of lessons throughout completing this project, including research skills, time and project management skills, and a large range of technical skills

10.1 Research

My research process was slightly different to what I expected. Due to the niche area that my project covers, there was a lack of literature surrounding the topic. There is a large amount of literature available regarding bots, but materials covering Discord, and especially Discord bots, are relatively scarce. Due to this, I resorted to using some materials that are not formally academic. Additionally, I found it difficult to pinpoint a precise 'gap' in the literature to fill, since the literature surrounding Discord bots is almost all gaps. Therefore, the justification for this project may not be as strong as for a project where a precise gap in the literature can be defined.

Aside from the research conducted on the literature surrounding the topic, I had to complete a huge amount of research into web development and the technologies I chose to use throughout the project.

10.2 Myself

10.2.1 Continued personal development

After completing the project, I identified several areas for continued personal development. These are my programming skills, software planning skills, time management skills, and general knowledge surrounding modern web technologies.

These areas were chosen partly because I believe web applications are the future of technology. From my own observations, an increasing number of technologies are becoming web based as programs are shifted from operating locally to operating in the browser or in the cloud. Therefore, improving my skills in these areas is very likely to prove beneficial in years to come.

I am choosing to continue my time management skills since this was one of the areas that I was lacking in throughout the project.

10.2.2 Skills

I found that the main skills required for my project were programming skills. Due to the large amount of programming with different technologies required for my project, I found having a solid knowledge of programming techniques to be very useful. Additionally, the ability to learn to use new technologies was very important; being able to read articles and documentation to learn new technologies was extremely beneficial to the project. Additionally, the limited time management skills I have proved useful, as getting into a routine and allocating regular, sufficient time to work on the project definitely aided in its completion. Traditional literature research skills were not as useful as anticipated but still proved quite beneficial. Due to the lack of literature surrounding the topic of the project, not a huge amount of research was able to be completed. However, some research of the literature had to be carried out of course, and so these skills were still important.

10.2.3 Strengths and weaknesses

I found my main strengths during the project to be my programming skills and my ability to learn new concepts quickly. Even though the code for the project was written in Node.js/Javascript, which is a language I don't have a huge amount of experience in, I found my general knowledge of programming techniques and concepts to be extremely useful. I was able to combine the skills I already had with the knowledge I gained throughout the project to produce what I believe is a good system. Most of my knowledge of React and Express was gained through the resources provided by Full Stack open (2021). My ability to follow this course and absorb the knowledge from it, along with being able to effectively utilize the documentation for the different technologies I used, allowed me to develop effectively using these previously unknown technologies.

I believe my main weakness was my time management skills. As stated in the Evaluation chapter, several features of the system had to be left out due to the time constraints of the project, and I believe the attempt at user testing failed partly because I distributed the questionnaire too late. During the project, I found myself spending time on features of the application that should not have been a priority. For example, I probably spent too long on styling the frontend of the application - this was not massively important for the project and so I could have achieved more by spending this time more wisely. Additionally, I spent a long time making sure the code for the system was clean - this was not entirely necessary and I believe there were several more important tasks that I should have completed instead of this. If my time managements skills were better and I was more efficient at prioritising my tasks, it is likely that beneficial additions could have been included in the project.

10.3 Topic

Throughout this project, I learned a great deal of skills in the topic of web development, bots, and Discord. These include:

- Developing a REST API
- Developing a moderately complex single-page application
- Using a NoSQL database
- Developing and publishing a package intended for use by other developers
- Developing a Discord bot
- Communicating with the Discord API
- Authorization using the OAuth2 protocol
- Deploying web applications
- Utilizing a virtual machine

I also learned how to effectively use many technologies, including React, Express, MongoDB, Node.js, npm, Postman, WebStorm IDE and DigitalOcean.

I learned how to do each of these through online research, primarily by reading articles and guides. I had little to no experience in any of these areas before beginning the project, and now I feel confident using them. I am very pleased with how much I have learned and I plan to keep learning for as long as I can.

Thank you.

References

Anicas, Mitchel (July 21, 2014). *An Introduction to OAuth 2.* DigitalOcean. URL: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2 (visited on 11/05/2021).

AnyforSoft (May 19, 2021). *10 Famous Websites Built with React JS.* AnyforSoft. URL: https://anyforsoft.com/blog/10-famous-websites-built-react-js/ (visited on 08/03/2021).

Arcane (2021). *Arcane.* URL: https://arcane.bot/ (visited on 11/03/2021).

Carey-Simos, George (July 6, 2020). *Discord Reaches 100M Monthly Active Users, Shifts Away From Gaming.* WeRSM - We are Social Media. Section: Featured. URL: https://wersm.com/discord-reaches-100m-monthly-active-users-shifts-away-from-gaming/ (visited on 11/03/2021).

Cho, Dana (July 15, 2021). *What the Fog.* Google Design. URL: https://design.google/library/what-the-fog/ (visited on 09/01/2021).

comScore (2014). *The U.S. Mobile App Report.* (Visited on 05/20/2021).

Curry, David (Sept. 9, 2020). *Discord Revenue and Usage Statistics (2021).* Business of Apps. URL: https://www.businessofapps.com/data/discord-statistics/ (visited on 08/06/2021).

Discord (June 25, 2016). *Discord - Free Voice and Text Chat for Gamers.* URL: https://web.archive.org/web/20160625141940/https://discordapp.com/ (visited on 11/05/2021).

— (2021a). *Discord — Your Place to Talk and Hang Out.* Discord. URL: https://discord.com/ (visited on 11/05/2021).

— (2021b). *Slash Commands.* Discord Developer Portal. URL: https://discord.com/developers/docs/interactions/application-commands#slash-commands (visited on 08/11/2021).

— (2021c). *State and Security.* Discord Developer Portal. URL: https://discord.com/developers/docs/topics/oauth2#state-and-security (visited on 08/31/2021).

Discord.js (Sept. 22, 2021). *Discord.js Guide.* URL: https://discordjs.guide/ (visited on 11/05/2021).

Dyno (2021). *Dyno - Discord platform.* URL: https://dyno.gg (visited on 11/03/2021).

Facebook (2021). *Introduction - Messenger Platform - Documentation.* Facebook for Developers. URL: https://developers.facebook.com/docs/messenger-platform/introduction/ (visited on 11/02/2021).

Full Stack open (2021). *Full stack open 2021.* URL: https://fullstackopen.com/en/ (visited on 11/05/2021).

Hardt, Ed (Oct. 2012). *The OAuth 2.0 Authorization Framework.* URL: https://datatracker.ietf.org/doc/html/rfc6749 (visited on 11/02/2021).

Henry, Shawn Lawton (Apr. 29, 2021). *Web Content Accessibility Guidelines (WCAG) Overview.* Web Accessibility Initiative (WAI). URL: https://www.w3.org/WAI/standards-guidelines/wcag/ (visited on 09/09/2021).

Howard, Philip N, Bence Kollanyi, and Samuel Woolley (Nov. 17, 2016). “Bots and Automation over Twitter during the U.S. Election”. In: *COMPROP Data Memo* 2016.4, pp. 1–5. URL: `http://geography.oii.ox.ac.uk/wp-content/uploads/sites/89/2016/11/Data-Memo-US-Election.pdf` (visited on 05/19/2021).

Jargon, Julie (June 11, 2019). “The Dark Side of Discord, Your Teen’s Favorite Chat App; Its private gaming communities can be like unsupervised playgrounds, full of racist memes, vulgar talk and bullying”. In: *Wall Street Journal (Online)*. Publisher: Dow Jones & Company Inc. URL: `https://www.proquest.com/docview/2237802537/citation/BCFB0BE3648848C7PQ/1` (visited on 11/05/2021).

— (Jan. 20, 2021). “Family & Tech: Discord App Aims for Safer Space”. In: *Wall Street Journal, Eastern edition*. Num Pages: A.12 Publisher: Dow Jones & Company Inc, A.12. ISSN: 00999660. URL: `https://www.proquest.com/docview/2478928880/citation/2ED89ED4F67040D9PQ/1` (visited on 11/05/2021).

Kiene, Charles and Benjamin Mako Hill (2020). “Who Uses Bots? A Statistical Analysis of Bot Usage in Moderation Teams”. In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–8. (Visited on 05/19/2021).

Klopfenstein, Lorenz Cuno et al. (June 10, 2017). “The Rise of Bots: A Survey of Conversational Interfaces, Patterns, and Paradigms”. In: *Proceedings of the 2017 Conference on Designing Interactive Systems*. DIS ’17: Designing Interactive Systems Conference 2017. Edinburgh United Kingdom: ACM, pp. 555–565. ISBN: 978-1-4503-4922-2. DOI: `10.1145/3064663.3064672`. URL: `https://dl.acm.org/doi/10.1145/3064663.3064672` (visited on 05/19/2021).

Lebeuf, Carlene, Margaret-Anne Storey, and Alexey Zagalsky (Jan. 2018). “Software Bots”. In: *IEEE Software* 35.1. Number: 1 Conference Name: IEEE Software, pp. 18–23. ISSN: 1937-4194. DOI: `10.1109/MS.2017.4541027`. URL: `https://ieeexplore-ieee-org.abc.cardiff.ac.uk/stamp/stamp.jsp?tp=&arnumber=8239928` (visited on 05/19/2021).

MinnDevelopment (May 15, 2020). *401: Unauthorized on /guilds/{guild.id}/channels - Issue #1008*. GitHub. URL: `https://github.com/discord/discord-api-docs/issues/1008` (visited on 09/10/2021).

Mock, Kenrick (Feb. 22, 2019). “Experiences using Discord as Platform for Online Tutoring and Building a CS Community”. In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. SIGCSE ’19. New York, NY, USA: Association for Computing Machinery, p. 1284. ISBN: 978-1-4503-5890-3. DOI: `10.1145/3287324.3293769`. URL: `https://doi.org/10.1145/3287324.3293769` (visited on 05/20/2021).

Mueller, Bernhard (Aug. 2, 2017). *Exploiting script injection flaws in ReactJS apps*. DailyJS. URL: `https://medium.com/dailyjs/exploiting-script-injection-flaws-in-reactjs-883fb1fe36c1` (visited on 08/31/2021).

Murgia, Madhumita (Nov. 15, 2015). “Can Facebook Messenger Kill Off Apps?” In: *The Telegraph*. URL: `www.telegraph.co.uk/technology/facebook/11996896/Can-Facebook-Messenger-kill-off-apps.html` (visited on 01/01/2018).

Pyltsyn, Alexey (Oct. 16, 2020). *Deployment — Create React App*. URL: `https://create-react-app.dev/docs/deployment` (visited on 11/04/2021).

React.js (2021a). *Accessibility – React*. URL: `https://reactjs.org/docs/accessibility.html` (visited on 09/09/2021).

— (2021b). *Components and Props – React*. URL: `https://reactjs.org/docs/components-and-props.html` (visited on 11/04/2021).

React.js (2021c). *JSX Prevents Injection Attacks*. URL: `https://reactjs.org/docs/introducing-jsx.html#jsx-prevents-injection-attacks` (visited on 08/31/2021).

Serby, Paul (Jan. 7, 2012). *Case study: How & why to build a consumer app with Node.js*. VentureBeat. URL: `https://venturebeat.com/2012/01/07/building-consumer-apps-with-node/` (visited on 11/03/2021).

surbhigupta et al. (Aug. 9, 2021). *Bots in Microsoft Teams*. URL: `https://docs.microsoft.com/en-us/microsoftteams/platform/bots/what-are-bots` (visited on 11/02/2021).

Top.gg (2021a). *Arcane*. Top.gg. URL: `https://top.gg/bot/437808476106784770` (visited on 11/05/2021).

— (2021b). *Discord Bot List*. URL: `https://top.gg/` (visited on 08/03/2021).

— (2021c). *Dyno*. Top.gg. URL: `https://top.gg/bot/155149108183695360` (visited on 11/05/2021).

Vladoiu, Monica and Zoran Constantinescu (Dec. 2020). "Learning During COVID-19 Pandemic: Online Education Community, Based on Discord". In: *2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet)*. 2020 19th RoEduNet Conference: Networking in Education and Research (RoEduNet). ISSN: 2247-5443, pp. 1–6. DOI: `10.1109/RoEduNet51892.2020.9324863`. URL: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9324863` (visited on 05/20/2021).

West, Jessamyn (Nov. 1, 2020). "Community via Discord". In: *Computers in libraries* 8, p. 11. URL: `http://web.a.ebscohost.com.abc.cardiff.ac.uk/ehost/detail/detail?vid=0&sid=94f8b2c1-c27c-43bc-bb22-88b1605c4ec3%40sdc-v-sessmgr02&bdata=JnNpdGU9ZWhvc3QtbGl2ZSZzY29wZT1zaXRl#AN=147057446&db=rzh` (visited on 05/18/2021).

Yaman, Julian (Sept. 7, 2021). *discord-bot-dashboard*. GitHub. URL: `https://github.com/shitcorp/discord-bot-dashboard` (visited on 11/03/2021).

Zakas, Nicholas C. (May 25, 2010). *Cross-domain Ajax with Cross-Origin Resource Sharing*. URL: `https://humanwhocodes.com/blog/2010/05/25/cross-domain-ajax-with-cross-origin-resource-sharing/` (visited on 11/04/2021).