Message Manager – Scholarship Report

Table Of Contents

Message Manager – Scholarship Report Supplementary Materials

Definitions

Discord: An instant chat messaging service separated into *channels*, *guilds,* and *messages*. Interacted with through *accounts* with some accounts being automated (*bots).* It provides a gateway and REST API to interact with automatically.

Guilds: (or servers) Separate collections of channels with a private invite only system for user accounts – many are managed by staff teams or moderators.

Channels: Separate places where messages can be sent to – messages are persistent.

Messages: A message sent by an account to a Discord channel.

Bots: Automated accounts

Webhooks: Method of externally sending an automated message to Discord without an account.

Embeds: A something that can be sent instead of, or alongside raw content on a discord message. They are used to send rich content, and originally started for link previews, and now their use is expanded. User accounts cannot send embeds, and bot accounts can.

Video Links

These are links to some videos which are referred to through the report.

send-example.mp4 https://youtu.be/DWykaR5jKz4 - Example of using the new send command

mobile-send-example.mp4 https://youtube.com/shorts/iOIDPn0nE5U?feature=share – Example of using the new send command on a mobile device

embed-edit-example.mp4 https://youtu.be/A50U-m2rPGQ - Example of using the edit feature for an embed

MVP-edit-example.mp4 https://youtu.be/PbJmWByefos - Example of using the edit command on the MVP version of the bot

Actions-example.mp4 https://youtu.be/5Uz1eAiSKqM - Example of actions

Message Manager – Scholarship Report Introduction

Description of project

The project is a Discord bot which provides ways to edit messages sent from the bot account in certain servers. This allows for more than one user to be able to update a single message – something that cannot be done in discord.

The Issue

As someone who maintains a global community Discord server, I often came across the issue that to update an informational message in the server, I would have to delete all the previous messages and resend them. These messages were static informational messages in read only channels (only staff could send messages), for the purpose of providing information to all users (for example server rules, or FAQs). I had to delete them because they had been sent by a different user, who had left the staff team or was not online when the message needed to be updated. This was an issue because it added more work to updating these messages, and it meant that for minor changes such as spelling, everything had to be redone.

I decided to investigate potential solutions to this issue, starting with solutions that may already exist.

Existing solutions

The main solution to this problem that I saw other servers using was using a shared Discord user account to send, and edit, these types of messages. The passwords to the account would then be shared amongst the staff team. The issue with this solution is that is still a hassle to log out of the current account, and log into the shared one, it causes security issues when a user leaves the staff team. It is also not allowed by Discord, and could potentially lead to action taken against that account, including deactivation. An account being deactivated would mean the messages sent from it would become inaccessible.

Other solutions were using webhook accounts to send / edit messages – these require a knowledge of the API and of webhooks to use them. This was the solution that I used for a while, but it was cumbersome to do custom API requests each time, and only I had the knowledge to do that in our staff team. Many Discord users also do not have this type of technical experience, making this option unavailable to most. To address this issue I decided to make a product with a user facing interface that did the same thing (managed the messages from a shared account) that could be used by all my staff team, and by other staff teams.

Task Description

Create a service that allows a set group of users in a server to carry out message actions on a message from a shared account (sending, editing and deleting). The

service will ensure that access to these actions can be customised by server staff, and the service must have a high-quality user interface and experience. The service must be able to be used by any server on discord.

Stakeholders

I've identified the different people that my product will affect, and how much consideration I need to give them.

Discord: Discord is a stakeholder because I will be using their platform and APIs to provide my service. They are affected as the service will have an impact on the platform for the users that use it, changing their experience in some way. To make sure I am taking them and their interests into consideration I will make sure to follow their terms of service [1] and guidelines [2], and their specific developer policy [3] and terms of service [4]. This will include following Discord guidelines around rate limiting, implementing caches were required to avoid placing Discord under stress, and implementing throttling mechanisms when traffic to the bot rises above rate limits.

Staff Users: (or any user directly using the service). These are the people who will be directly interacting with my service. They will be affected by my design choices, the interface of the service, and the timelines for releases. To ensure I am meeting their need I will go to these users for the most feedback – and take their feedback into account, they will be able to give the most valuable feedback on actually using the service, as they will be using it. I will create a support server, a server specifically for users to come to for help, bugs, or feedback. This bot will include links to the support server.

Other Users: Who will be seeing the messages that are sent by above users. They are affected by the presentation of the messages, and the content of the messages. I will have to take them into consideration by ensuring that the content of the messages is not harmful.

Message Manager – Scholarship Report
Minimum Viable Product

As this was going to be a significantly large project, in terms of resources and time, I first decided to make a version that would meet minimum requirements, and release this to a subset of users. This is to explore potential solutions, and to explore how much of a demand there is for this.

Specifications

For the MVP I've trimmed down the overall specifications for the product to cover just the bare essentials

The service will:

- Allow Staff Users to send, edit and delete messages
- Allow Staff Users to have some control over who can do the above using a permissions system

These specifications exclude the specification of having a "high quality user interface" – so that it can be created quickly.

Research – Before Development

Language

There are two main options for me for languages for building the MVP in, Python or JavaScript.

Python

Python is a high-level programming language with a easier to learn syntax, it has a large user base, and is often an entry level language. Python has a community-maintained library for interacting with Discord's API, discord.py [5] which makes it easier to start off as I have not had prior experience with interacting with raw API's and gateways. The library also provides abstractions and tools for developing complex bots, such as cogs [6] which make it easier to separate out code into a logical structure, and the commands extension [7] which handles command parsing. I have also had experience with python through class and self-learning.

JavaScript

JavaScript is a very popular scripting language used by many developers, it usually runs in the browser, but also has a server-based engine, Node.js, which I would use for this project. There is an extensive community-maintained library for interacting with Discord's API, discord.js [8] which also has a commands extension. I have not had much experience with JavaScript and Node.js.

Message Manager – Scholarship Report

Decision

I have decided to go with Python and discord.py, because discord.py seems like a well maintained and documented library, and I have already had experience with Python, saving learning time compared to JavaScript.

Database

I will need to store data about each guild to store configuration, like access configuration. Later, after the MVP I will likely expand what data I will need to store, so it's a good idea to get familiar with one database through this project. Therefore, when choosing the database, I will take into consideration future proofing.

NoSQL vs SQL

Either NoSQL or SQL is an option here, SQL is the traditional, more structured database type, better for relational data that doesn't change its schema. Whereas NoSQL is typically document / object based and suited for more dynamic data.

I have chosen to go with a SQL database as the data I will be working with will be very related. This is because most of the data will be linked with higher level entities, like the channel or the guild, this is for potential future improvements. The data is also not dynamic, as any changes to the schema will happen as a release and this can be managed through migrations.

I have decided to use PostgreSQL as the SQL database, as it is open source and free to use, has a good amount of support in python, and runs on most operating systems.

Development Tools

I will use git as a versioning system, to track changes and to provide a structed development flow. I will use VSCode as my editor, as it suites my project and has tools for python, and it is free, it's also pretty light so I don't need to worry about being able to run it. For python package management I will use PipEnv to manage them, this is so I don't have an issue with dependency mismatch between development and production environments.

Development

The first aspect that I will develop is the core feature of sending, editing and deleting messages. I will implement permission gating for these commands after that, and then any other improvements that are needed. I will not be able to release it to users until permission gating has been implemented.

User interaction flow

Users will interact with the MVP through message commands, these are messages sent to a channel in Discord

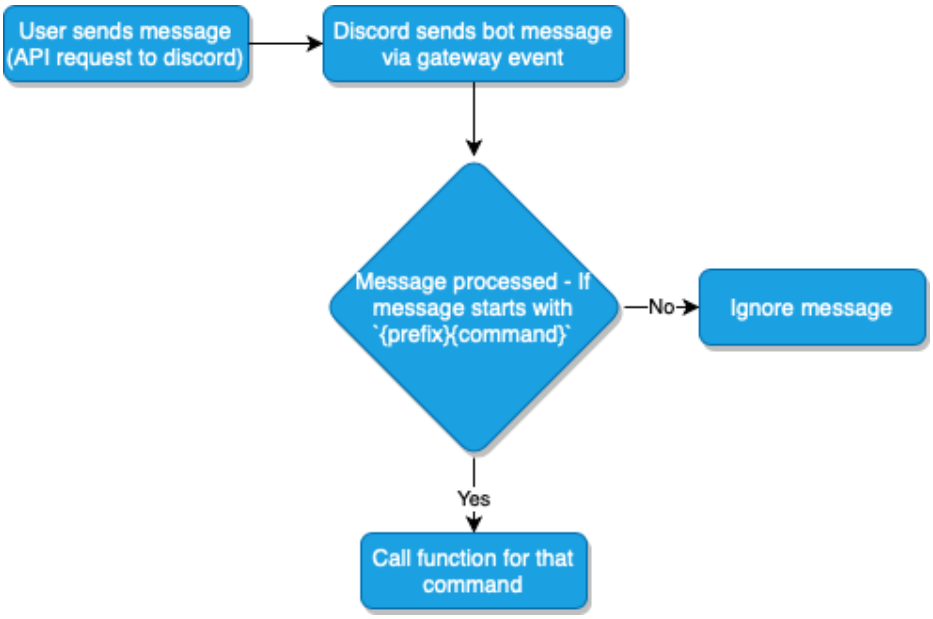


Figure 1 - Diagram of flow of information

This diagram (fig 1) represents the flow of information when a command is used – this is the typical flow for message-based prefix command Discord bots.

I've decided to use "~" as the prefix, as it's memorable and is unlikely to clash with other bot's prefixes.

Directory layout – and separation of code

I've setup the files in a way that will hopefully be useful for the future and to allow ease of code separation to prevent too complex files.

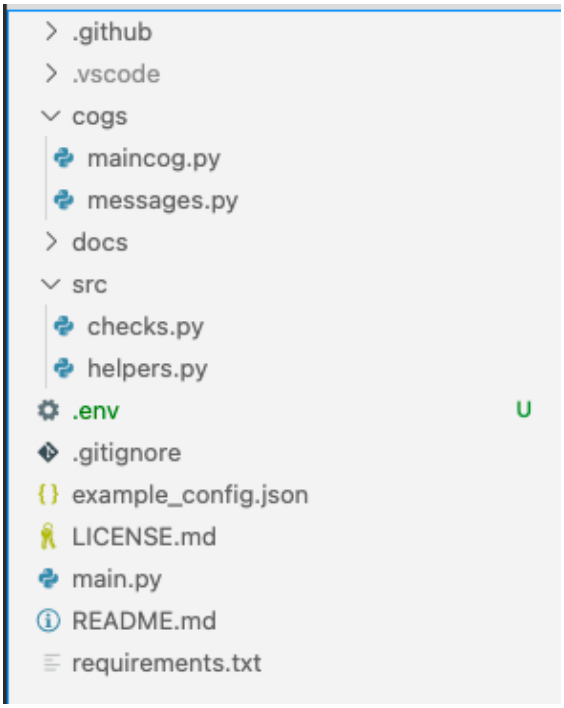


Figure 2 - Directory layout

main.py (fig 2) is the entry-point file, this is where the bot instance (instance of a discord.py class) is created, and where config is setup, and cogs are loaded.

Cogs are separate command logic groupings, they use the discord.py Cog class, and can be reloaded on demand, allowing for refreshing the code without restarting the entire bot. maincog.py has various information commands (like ~ping for checking online status), and message.py has the code to send, edit and delete messages.

/src contains various functions that are reused, like bot admin checks (checks.py), and embed creation (helpers.py).

Message Management Code

```
16      @commands.command(name="send", rest_is_raw = True)
17      async def send(self, ctx, channel_id=None, *, content=None):
18          try:
19              await ctx.message.delete()
20          except discord.errors.Forbidden:
21              pass
22          channel_id = await helpers.check_channel_id(ctx, channel_id, self.bot)
23          if channel_id == False:
24              return None
25          content = await helpers.check_content(ctx, content, self.bot)
26          if content == False:
27              return None
28          if content[1:4] == '```'and content[-3:] == '```':
29              content = content[4:-3]
30          channel = self.bot.get_channel(int(channel_id)) # Get the channel.
31          msg = await channel.send(content)
32          await helpers.send_message_info_embed(ctx, 'Send', ctx.author, content, msg)
```

Figure 3 - Send command code

Figure 3 shows the initial code for sending a message via the bot. Line 18-21 are to delete the invoking command, with handling if that was not possible. Line 22-30 involve checking the inputs for validity and fetching the discord.py instances of channels. Line 31 sends the message to the channel – and 32 sends the success message to the user. To get to this point I used an iterative process, starting with a command that didn't have line 18-29, and then later realised that checks on the content were needed as errors occurred.

Database Implementation & Custom config

During the testing and initial development of the message management feature, I just used a simple check against a statically set variable (in a JSON file) for permission management. This means it will be only able to work for one guild as the other guilds cannot set their own permission roles. Also to be able to work for other guilds, staff members of those guilds must be able to set their config through the bot, as it is too cumbersome for me to add a value to the config file every time the bot is added to the guild.

So at this stage the database was required.

In addition to storing the management role ID for permissions, I also decided to allow the customisation of the prefix for each guild.

```
create_db = """CREATE TABLE IF NOT EXISTS servers (
    server_id bigint NOT NULL UNIQUE,
    management_role_id bigint,
    prefix VARCHAR(3) DEFAULT '~',
    bot_channel bigint,
    member_channel bigint
);"""
```

Figure 4 - Database Schema

Figure 4 is database schema, all the IDs had to be Begins as Discord IDs are bigger than the greatest integer.

In the database schema, the prefix has a default value so that as soon as the server entry is created (which happens when a command is used), the default prefix will be stored. This means if the default prefix changes at any time only new servers after the change will be affected.

I then implemented a basic check for management_role_id on all message related commands, and a set of commands allowing the config values to be customised.

I've used PostgreSQL (as the prior decision), and asyncpg for the library.

Issue with asynchronous code

```
async def _create_tables(self):
    conn = await self.pool.acquire()
    await conn.execute(create_db)
    await self.pool.release(conn)
```

```
28
29  async def _create_tables(self):
30+     async with self.pool.acquire() as conn:
31+         await conn.execute(create_db)
32
```

Figure 5 - Async changes (git diff)

Figure 5 shows the changes for fixing a potential race condition, that was having an effect when long running connections were used, which was slowing down the bot. The original method resulted in pool connections not being released correctly if an error occurred in the execution (which for some create commands was expected, and handled). This led to a lot of unused connections taking up memory. The async with method meant that whatever happened, the connection would be released once the code inside the block finished executing, including if an error was raised.

(Note this is just a small example, this change was repeated across all database functions.)

Message Manager – Scholarship Report

Code Formatting

Something I had not considered in the initial planning was a consistent code style. Throughout all the code I followed the pep8 python style conventions, but this was not enough and still allowed for some messy or not easy to read code, which became more of a problem as the code base increased. So I decided to look into a tool for formatting and chose the Black style formatter, which applied strict styles across all the code. This helped with later readability when coming back to code.

Embeds

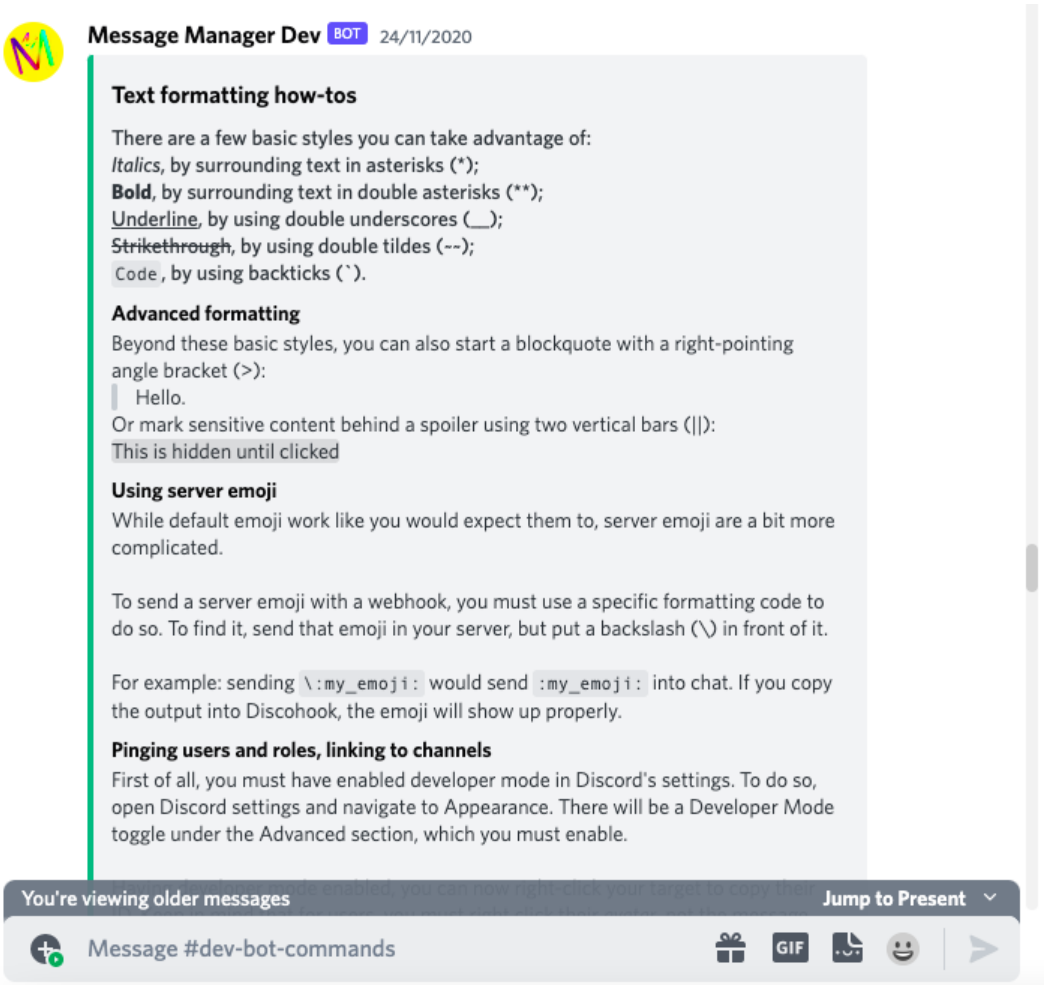


Figure 6 - an example of a Discord embed

Embeds are a way of styling content to some degree in Discord messages, but embeds can only be sent by bot accounts. After I finished implementing sending, I decided to added embed sending, editing and deleting commands as well so - and other users - could take advantage of being able to use embeds. This involved adding embed-specific

commands, and working out a JSON parser.
The input from the users for the embed commands is JSON.

Figure 7 - Embed sending example command

This is not user friendly as the user must understand JSON, the required format, and get the format correct with only a text box. But there was no other way to implement embed management, at least any other ways that were viable for the MVP.

This is something I wish to improve in the next version, ideally a UI where the user sets each field separately. However, this was a feature I wanted to include as it was very useful, and this was eventually shown by the embed command usage being high (over 500 total runs). To mitigate the effect of the bad UI I included links to embed code generator website in the bot and docs to assist users in using it.

Message Manager – Scholarship Report

User response

After finishing some final touches to the MVP I released it to all Discord users. This involved placing it on bot discovery lists, which are websites people have setup to allow Discord users to find bots that suit their needs.

The MVP was a success, with many really liking the bot and what it does, this is partly shown by (at the time before the next version) the bot was in nearly 600 servers, that means 600 staff teams / owners of servers added the bot to their server, and then did not remove it, this compared to a total of 1758 server it had even been added too, which is about a 33% retain rate.

I also got positive feedback from users in the support server I had setup for the bot.

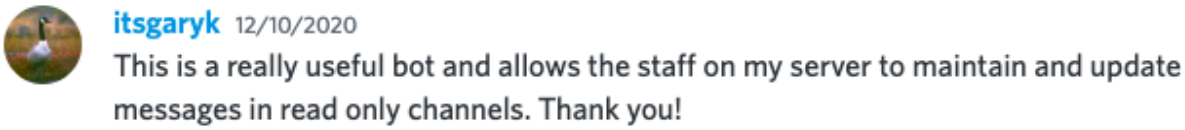


Figure 8 - Positive feedback from GaryK

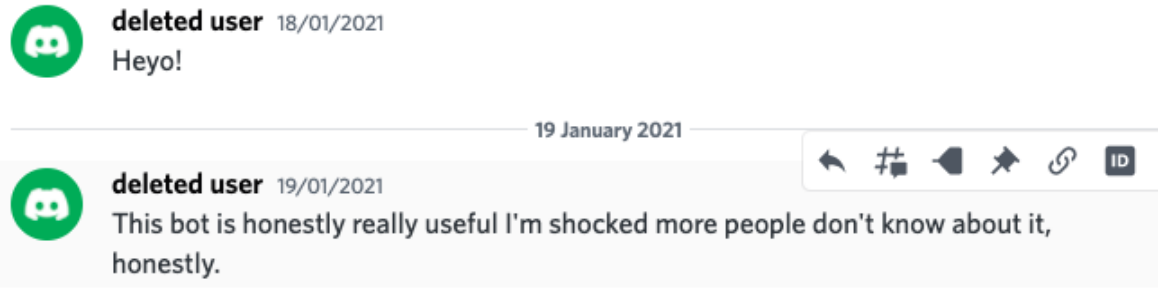


Figure 9 - Another example of positive feedback

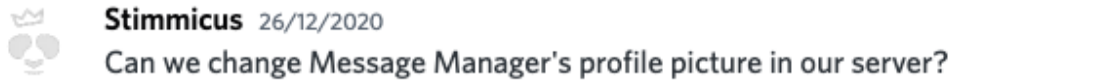


Figure 10 - A feature request / question from a user

The profile picture is the bot's picture displayed next to the image. I'm unlikely to implement this as I do not see a way to do this well at this point in time, considering the Discord features that are available to me. This is not great, if I could add this as a feature it would improve the look and customisability of the bot.

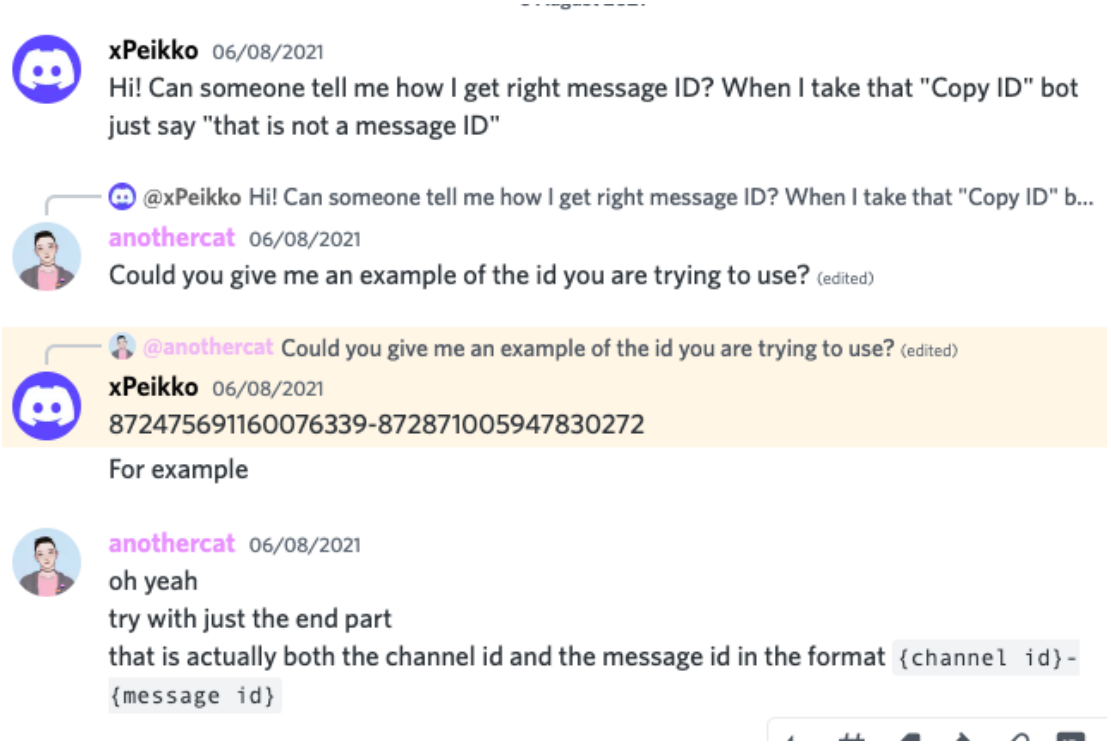


Figure 11 - Reported issue with editing a message

This user came to me with the issue that the bot wasn't working when trying to edit a message (fig 11), the command they are trying to use is has an argument for the message ID, to identify which message to edit.

The real issue here was that Discord allows for two ways of copying a message's ID, one way which just copies the ID and another which copies both the channel ID and the message ID with a "–" in between. The bot was rejecting the extended version as it did not recognise it.

While this wasn't a bug, and the bot was working as intended, this was something that I could see could cause confusion and happen often. So I edited the argument parsing

functions to accept both formats, and to extract the message id from the extended ID if that was provided by the user.

Summary of the product and how went

See (MVP-edit-example.mp4) for a video example of editing a message. This video was taken in the bot support server.

The MVP has shown me that this is something there is significant demand for, is something that can be implemented well within the confines of discord, and has given me clear path forwards. To move forwards I will focus on improving the UI of the core message editing commands, and improving the overall usability of the bot.
I did not come across too many issues during this process, but did receive some feedback on future features.
I will now investigate the next steps to fulfil the full specification.

Message Manager – Scholarship Report
Final Outcome

This is the next stage after the MVP, during this stage I will use the MVP as an example and start point, and move forwards by improving upon it.

Intro

Specifications

This is the same as the Specifications from the MVP, but extended.

The service will:

- Allow Staff Users to send, edit and delete messages
- Allow Staff Users to have some control over who can do the above using a permissions system
- Provide an intuitive user interface and flow – especially for new users
- Documentation
- Require the least technical know-how from a user as is possible

I've added the last requirement, which wasn't something I was considering at the start of the project, because when I implemented the embed system it required knowledge of JSON, which puts the barrier too high for new users.

Changes to the Ecosystem

Since I've started the project many things have changed with Discord, and the programming ecosystem around it.

Firstly Discord have introduced interactions, including slash commands [9], which are a much more user friendly way of interacting with a Discord bot compared to message commands (explained in MVP). These include a command auto-complete, inbuilt buttons, select menus, and context menus on messages. I intend to use these features to provide a better UI for users.

Interactions also can be received via outgoing HTTP webhook, which is something to consider.

With the release of interactions, Discord have also decided to make message content, for all messages not sent by the bot making the API request, hidden unless approval from Discord is gained. From the information of the approval process that was released, I was not going to be eligible for approval, making the choice to move to interactions mandatory as this change broke message based commands.

Secondly, discord.py, the Python library I was using for interacting with Discord was discontinued, and there are no other high quality Python libraries at this stage. This poses an issue to me as it meant I either had to move away from Python, use a differently lower quality library, or make my own Python library.

Message Manager – Scholarship Report

My Vison of the Product – Non Programming view

This is a breakdown of interpreting the specifications for each feature.

- Sending a message should be quick, and easy, and available as soon as the bot is added to a server.
- Editing a message shouldn't require copying and pasting multiple ID's, and the content for editing the message should be pre filled (in the MVP it was just replaced by what they sent), so that a small edit is easy.
- Editing and sending embeds should be easy to do, and not require JSON. All embed fields should also be validated before sending to avoid the action getting rejected at the last step.
- Permissions should be easy to setup, but highly customisable, to the level that Discord permissions are. This means that both a new user, and a user who knows the bot well will be able to take full control to their needs. Permissions should be granular for each type of action, and have sensible defaults.
- Eventually all actions should be able to be taken from a web based dashboard, as well as through discord. This is because it is possible to deliver a much better interface via the web, while making it quick to use through discord. However, the through Discord interface should be the priority as this is what users expect.

Research – Before Development

Stack

This is the big picture of how the bot will work, this might have to change as it is no longer possible to stay with discord.py

I've explored some potential options that I could take:

Using a discord.py extension - continue with Python

This would use a library I've found out about that monkey-patches, which is here editing another libraries' code at runtime, discord.py to support interactions and for any bug fixes. I would use this with discord.py, and then possibly build an IPC (Inter-process communication) connection to a Python web server for a web API to serve the website. This would be the easiest in terms of continuity, I would be able to keep most of my codebase from the MVP, but be fairly complicated in terms of the IPC, and be bug prone (I've already found significant bugs with the extension, which is maintained by beginners to the situation like me).

Transition to a new language – and use another Discord API wrapper

This would mean using a new language (most likely JavaScript) because there is not a well maintained Python library, but still having the same bot structure with a gateway connection for all bot events. This would require a full rewrite from the MVP, learning a new language / library and also still require an IPC or similar connection for a web API. However it would be less bug prone as the library would be maintained by a large amount of experienced people.

Message Manager – Scholarship Report

Implement a custom solution

This would mean a full rewrite, and building a custom library for interacting with discord's REST API and gateway services. It is language agonistic, as it's custom and can be implemented in any language. This would require a lot of work, and rely solely on the quality of my code for complicated stuff that was previously handled by discord.py. If I chose this option, I would build it as an API service, receiving HTTP incoming webhooks for interactions instead of over the gateway. It would however require a gateway instance to provide a shared cache for some important data that I cannot fetch all the time (due to Discord rate limits), such as guild and channel data). I would be able to slim down the gateway connection. This would make it much easier to implement an API for a web interface, however it would require a lot of work.

Decision

I've decided to go with the last option. This is because:

- I've lost trust in current large maintained libraries and don't want to face this much of a change again. I feel like it is better to do the hard work of a large change now so I don't have to in the future.
- It will be much less resource intensive – especially as the gateway will be able to be slimmed down a lot. Most libraries do not allow for much customisation, and use up a lot of memory in caching a lot of things, many that I do not need to be cached.
- It will allow me to easily implement a web interface.

However it will require a lot of work as it is a complete rewrite. In the meantime I will keep the MVP running, and complete bug fixes on it as they appear so as to not interrupt service.

Language

As I've decided on a complete rewrite, there are no benefits from remaining with python because of the codebase being in python. Therefore I'm reconsidering what language I will choose.
I've decided to use JavaScript, with TypeScript for type security. This is because I've now learnt JavaScript, and it's slightly more mature for API development. There is also the benefit of sharing code in the future with the web interface.

Library and Services choices

Now that I have a direction in terms of stack and language I can decide what libraries and other services to use.
I will require:

- Database
  I've decided to stick with PostgreSQL – I've had no issues and see no reason to change
- Database ORM

As I've changed languages I can no longer use the library / ORM I was using – I've decided to use Prisma, as it has type-safety for TypeScript built-in.

- A web server
  For this there are multiple options, I've narrowed it down to two:
  - Express – A very popular web server, used by many.
  - Fastify – Express-like in terms of how it is used, but also includes type safety and a good plugin system.

  I've decided to go with Fastify as it's got slightly better features for me.
- A Cache – I've decided to use Redis – it's widely used and fast.

A note on type safety

I've taken type safety into consideration in a lot of the previous decisions, and this is because I find it extremely useful when developing a complex project. Type safety and extensive typing is something that is becoming common practice in many places, and it catches a lot of bugs that would otherwise go unnoticed.

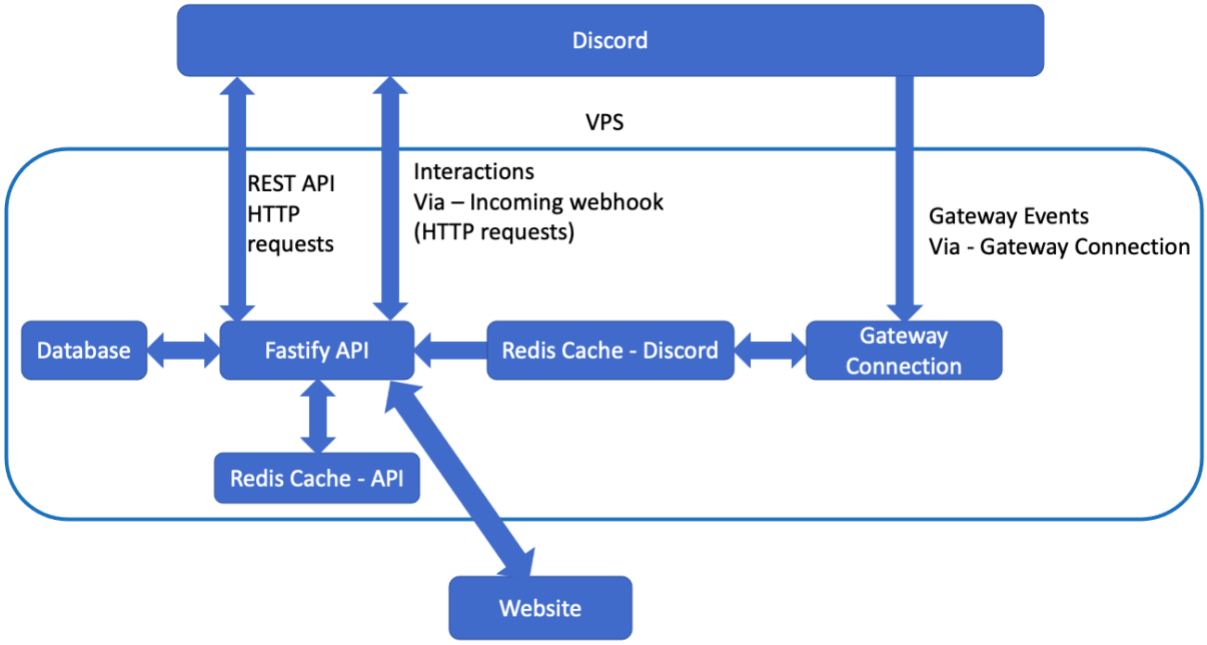
Development

I've decided to break the approach to development into two stages, first the gateway connection and cache, and then the API and overall service. This is because they are distinctly different approaches to each.

Message Manager – Scholarship Report

Information Flow

A Diagram showing the information flow between the services planned. A single direction arrow indicates information which goes in one direction (this does not count GET or similar only type requests as a flow of information), and two signifies information both ways. All boxes represent a service, and those inside the “VPS” box are the ones which will run together on a server. The website is included as it could happen in a future plan.



12 - A diagram of the information flow of the new stack

Gateway Cache Development

The purpose of the cache is to cache objects that cannot be fetched on-demand due to discord’s rate-limits. These were: Guilds, Channels, and the Bot member object. For permission checks, names, icons, etc. Discord sends events down a gateway connection, which is built off WebSocket’s, when related Discord entities change.

The cache uses Redis as an in-between store, that can be accessed from different JavaScript process, and an instance of the code that I built to interact with this cache would run on both the API server, and the Gateway Connection. I chose this method as it was simpler than attempting to run both a webserver and a gateway WebSocket connection on one process.

My approach to this was to look towards other libraries for their implementations of a cache. After some research I eventually decided to keep using a library to handle the gateway connection – as this was a very low level things that I did not have the experience to approach correctly – and use it’s event handlers to implement the cache. I chose detritus, a typescript client built for simplicity and speed for this.

I initially used only their base gateway client, but this led to events failing to be processed after a reconnect had to happen (for example when there was a minor internet issue), so I moved to their full client and disabled things like in-memory cache that was included in their client.

```
30
    You, 4 months ago | 2 authors (Sam and others)
31  export default class Guild extends BaseStructure<
32    Snowflake,
33    CachedMinimalGuild
34  > {
35    _structureName = _structureName;
36    constructor(id: Snowflake, { redis }: { redis: ReJSONCommands }) {
37      super(id, { redis });
38    }
39    static saveNewUnavailable(
40      data: { id: Snowflake; unavailable: boolean },
41      { redis }: { redis: ReJSONCommands }
42    ) {
43      const strippedDownData = { unavailable: data.unavailable };
44      return this._baseSave(strippedDownData, data.id, {
45        redis,
46        structureName: _structureName,
47      });
48    }
49    static saveNew(
50      data: GatewayGuildCreateDispatchData | APIGuild,
51      { redis, client }: { redis: ReJSONCommands; client: GatewayClient }
52    ) {
53      return this._baseSave(
54        parseGuildData(
55          data,
56          client.clientId! // Must be set as READY event was received      Sam,
57        ),
58        data.id,
59        {
60          redis,
61          structureName: _structureName,
62        }
63      );
64    }
65
66    overwrite(data: CachedMinimalGuild): Promise<void> {
67      return Guild._baseSave(data, this.id, {
68        redis: this._redis,
69        structureName: _structureName,
70      });
71    }
```

Figure 13 - A section of the Guild Structure class

For the cache I used the technique of Structures, where each Discord entity or object is represented by its own class. Each structure has a constructor, methods to save new instances, methods to update and to get attributes of the structure.

Since the cache is in Redis, and gateway events may only include the updated field, the instance is only initialised with an ID and the Redis client, all other operations are done via a Redis call. This also means that instances of structures can be discarded quickly, allowing for low memory (RAM) usage.

These structures are also used by the API server's code – to access the cache.

While I was working on the API server, I discovered a performance issue with this design which was there were a lot of checks that checked similar fields on the same guild instance, and so many Redis commands were being repeated, which was because while Redis is fast it had a significant impact on the speed of a command running. To solve this I implemented an in-memory cache, on the level of commands, which would be discarded after that command / request was over. While this may seem like it is unintuitive as it's double caching, the in-memory cache is much faster for repeated commands, and it was both simpler and faster than any other solution, and had little impact on memory usage as the in-memory cache is discarded so fast.

Once I had finished working on this, I released it as a publicly available library, which someone contacted me about to use. I then used that library in my API server's code.

API server – Interactions

The Fastify web server will receive interaction webhook requests from Discord to the /interactions POST endpoint, this will happen whenever a user uses an interaction.

Security

Something that must be considered when receiving incoming HTTP requests is security, especially here as the input has to be considered trusted, as it can be used to take editing / destructive actions by a user. Discord has addressed this by signing all requests – and I implemented a secure check using the in-built crypto module. This is important to ensure is working correctly, otherwise a malicious actor could send fake interactions and pose as other users and take actions on their behalf.

Another aspect where security comes into play is the server itself, and DDoS and other attacks. To mitigate this I've hidden my IP of the server behind Cloudflare, and used a Cloudflare tunnel to tunnel requests directly to the API server, meaning all incoming requests can be blocked. [10] This is safer as attackers have to get through Cloudflare first, which is much harder. It also means I can spend less time worrying about security.

Another potential issue with security is data theft, and to mitigate this I've encrypted all sensitive database fields (such as message content, etc).

Message Manager – Scholarship Report

Interaction Handling

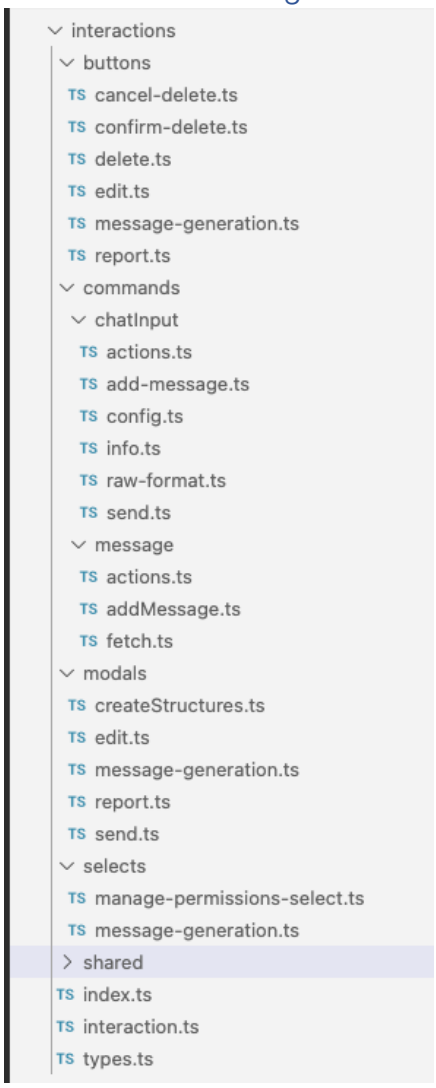


Figure 15 - Layout of interaction code

Figure 14 - Collapsed interaction handler

Figure 15 is the layout of how the code for interactions was organised – each type of interaction separated into a folder and each command / button / etc having its own file. (The example of the files is the present day version, to show the eventual organisation)

All the interaction requests go to a handler in index.ts (shown in fig 14), which then calls a function on the handler (to the left), which depending on type, name, custom_id then calls the correct logic to handle it.
This is required as there are so many different commands and interactions that could be called for this product, that it would become too complicated to have them combined.

Message Manager – Scholarship Report

Message Sending, Editing and Deleting

After the interaction system was setup, I started work on the message management function. Working with the new interactions was hard initially, as it required more than one interaction to complete an action. (for example sending was two interactions, the first a command and the second a form (modal) response). This required careful planning around state.
When making this function, I decided to separate out all the logic that actually did the message actions from the interaction handling code. This would allow for that logic to be reused at a later date for a web interface. It also improved the DRY-ness of the code.

One specific example of how I implemented it is sending. For sending a command to send a message to a certain channel is received – and then permission checks are done – and then the user is sent a modal to fill in with the content. When this modal is received (as another interaction), permission checks are redone, and then the message is sent.

See send-example.mp4 for a video example of the process for the user.

I also implemented a confirmation check on the deletion process, to double check if the user wanted to carry through with this action, as it's a destructive process. This was something that was asked for by users.

Actions

Actions were a concept I came up with to enclose editing, deleting and reporting a message. The idea with actions is it is one command, which response with buttons for the three, but only if the user has permission to carry out those actions. (for example if the user only had permission to edit, only the edit button would show up). This makes it easier for the users, as there isn't a command there that they cannot use.

See actions-example.mp4 for a video example of this.

Mobile

It was previously very hard for mobile users to use this product with the MVP, as they would have to get a lot of text in the right order in a small text box, but with interactions it makes it a lot easier. This is important because many Discord users only use mobile, and it was something that was requested by users.

See the mobile-send-example.mp4 video for an example

Permissions

The next thing to tackle was permissions, which I also intended to give a major overhaul to improve. The MVP worked by having a single role per guild which would allow access to all management commands, which worked but was limiting and I had users ask for that.

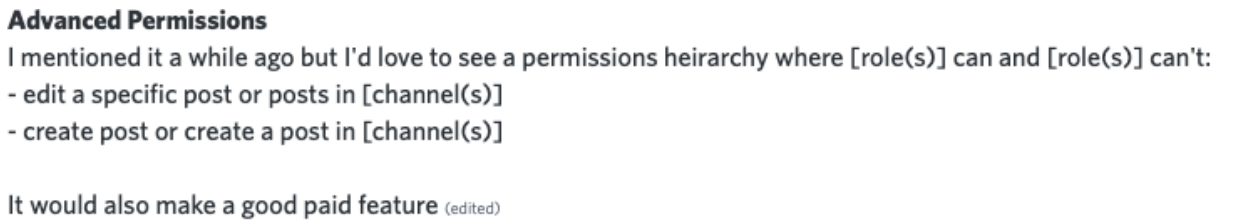


Figure 16 - Feedback on permissions

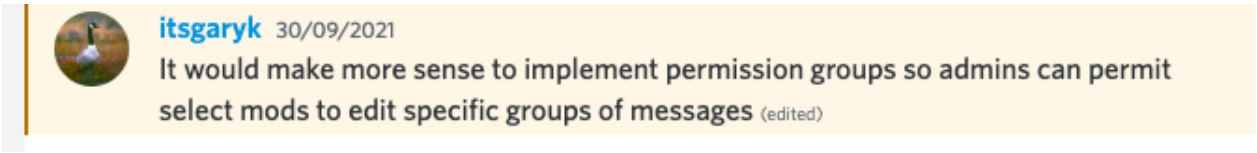


Figure 17 - Feedback on permissions

I decided to follow discord's permission system as an example. I couldn't use their permission system, and have staff assign their users Discord permissions for my service, as none of the actions on my service corresponded to Discord permissions. However I used their structure and format, and a big reason for this is that many will be used to this.

The system that I eventually designed used bitfields flags, which allow for permission values of on or off to be condensed into a short number, which is useful for storing. I also had different permissions for different actions, so a different permission for editing, sending, deleting, managing permissions, and managing config, allowing for finer control. I also included overrides for users and channels, meaning that a permission could be allowed for a user overall on the server, but denied in a specific channel. (for example a sensitive channel).

```
6
7   Permissions are stored in integer bitfields.
8   They're stored in a object in a column in the database.
9
10  Server level roles just have an allow bitfield, whereas server level user, and channel role + user permissions have both allow and deny.
11
12  eg
13
14  Server:
15  {
16  "roles": {roleId: bitfield},
17  "users": {userId: {allow: bitfield, deny: bitfield}},
18  }
19
20  channel:
21  {
22  "roles": {roleId: {allow: bitfield, deny: bitfield}},
23  "users": {userId: {allow: bitfield, deny: bitfield}},
24  }
25
26  Allow takes precedence over deny.
27
28  The flow for working out permissions is with bitwise OR:
29
30  (in a channel)
31
32  1. All role permissions for the user combined
33  2. User deny
34  3. User allow
35  4. Channel role permissions deny
36  5. Channel role permissions allow
37  6. Channel user permissions deny
38  7. Channel user permissions allow
39
40  (at the server level)
41
42  1. All role permissions for the user combined
43  2. User deny
44  3. User allow
45
46  Permissions checked using bitwise AND
47
48  ## Different levels
49
50  On different levels there are three states, either inexplicit inherit, explicit deny and explicit allow
51  Which allows a general approach, with small exceptions
52  For example:
53  A user has roles which allow send, edit
54  On all channels without an explicit permission set for that user / that user's roles (in allow or deny) that user has send, edit
55  In channel A one of the user's roles allows delete, in that channel the user has send, edit, delete
56  In channel B one of the user's roles denies send, edit - in that channel the user has no permissions
57  On all other channels the user has send, edit
58
```

Figure 18 - Internal Docs for permissions

As this permissions system was complicated, and very critical it doesn't contain false negatives or positives for security reasons, I wrote myself docs (a point of truth), and wrote some automates tests for permissions logic, which did flag some bugs.

However, it still needed a way for users to manage these permissions, for this I used select menus.

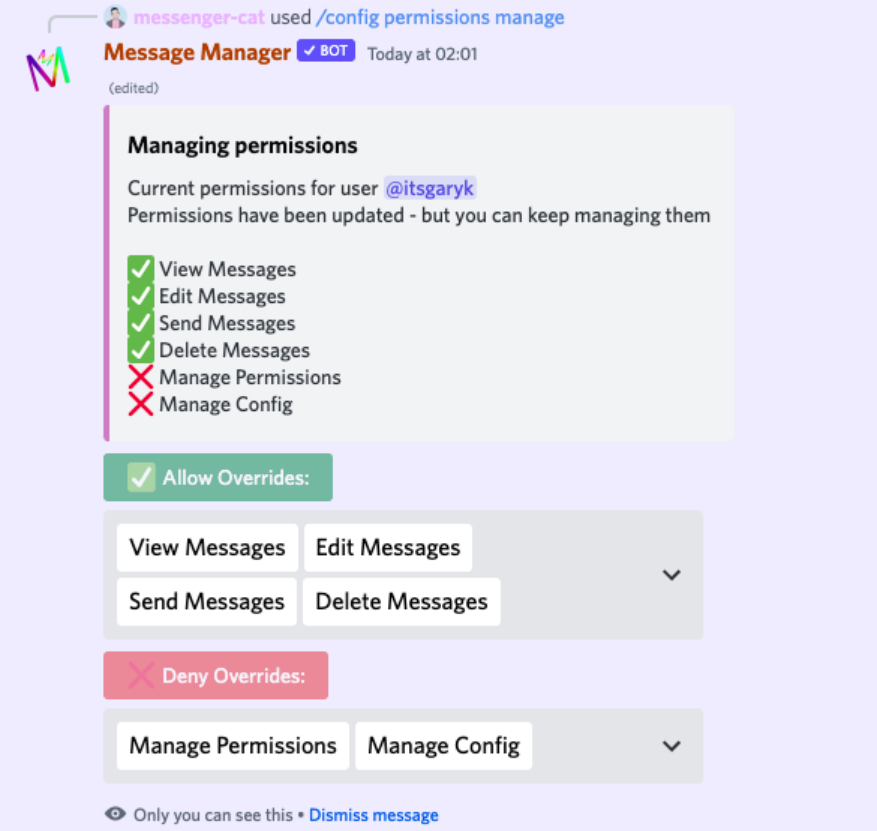


Figure 19 - Permissions management example

Unfortunately, Discord doesn't currently have inline toggles for bots, so I was unable to build the editing directly into the embed, however by using the select menus in this way I've managed to provide a clear design.

However, this wasn't simple enough for beginners, and this was pointed out to me when I asked for feedback on the system. (TODO: Show feedback)

As a result of this feedback, I created a command for permissions QuickStart, which has two pre-sets, one which allows all message permissions, and one which allows all management permissions. With this, new users can just run the command once, and it'll work for whoever they set in the command. I also wrote a section in the docs called QuickStart, with a guide on how to do this.

I also expanded the permissions design and used more emojis – for a more visual effect, easier on users.

Message Manager – Scholarship Report

Embeds

While embeds were included in the MVP, they were in a non-user-friendly state. I didn't include them in the initial development of this stage, as it was a significant amount of work, and I wanted to get the base work completed before I attempted this.

My plan for embeds was to use buttons to have different "screens" of editing, and some buttons leading to modals for content input. This would mean each field of the embed would be presented in a workable state, instead of having to give JSON of the entire embed. While editing / building the embed it doesn't change any messages, a cache is just updated, this is to prevent unnecessary changes and API calls. When the user is done, they press the "edit" or "send" button, which makes those changes on the corresponding message / sends it.

As raw text can be included alongside embeds, I also added an editing raw content button, to the initial screen. The embed flow was added to the sending / editing commands instead of separate commands. This reduces complexity.

See embed-edit-example.mp4 for a video example

Documentation

To assist the users in using the bot I have also made a website with detailed guides on all the bot's functions. While it should be easy to figure out how to use the functions, documentation helps speed that process up and assists those who don't understand it right away.

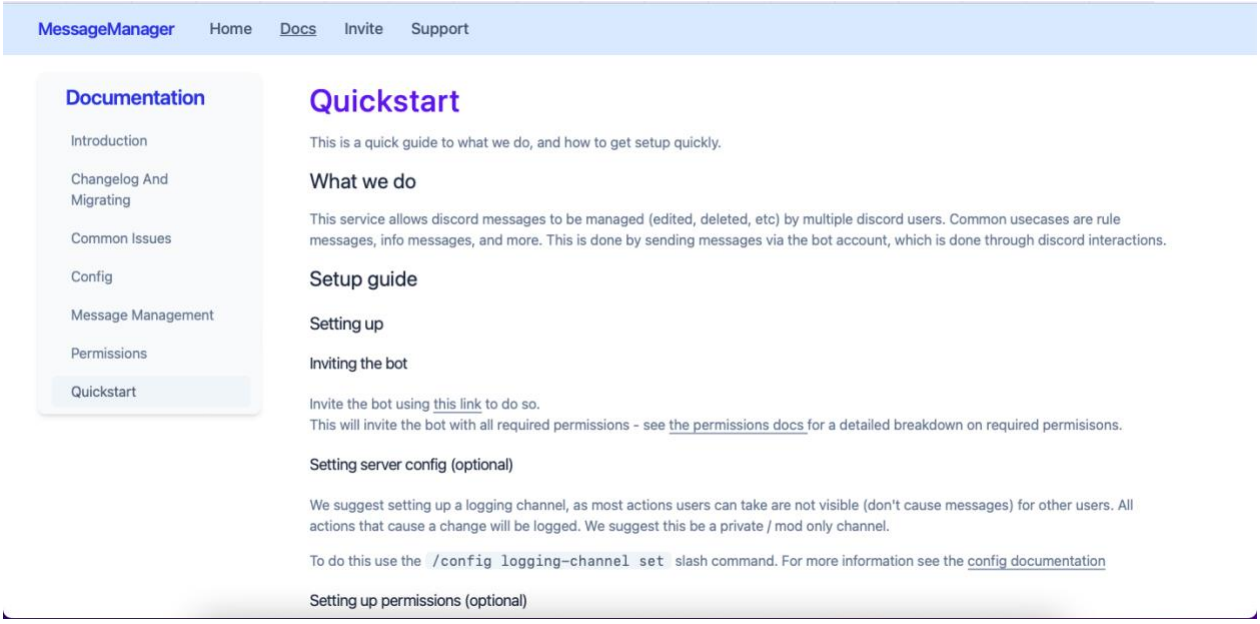


Figure 20 - QuickStart Documentation

Message Manager – Scholarship Report

Testing & Trailing

To ensure that the product worked and worked in the ways users expected I asked some users to test the beta version of this.

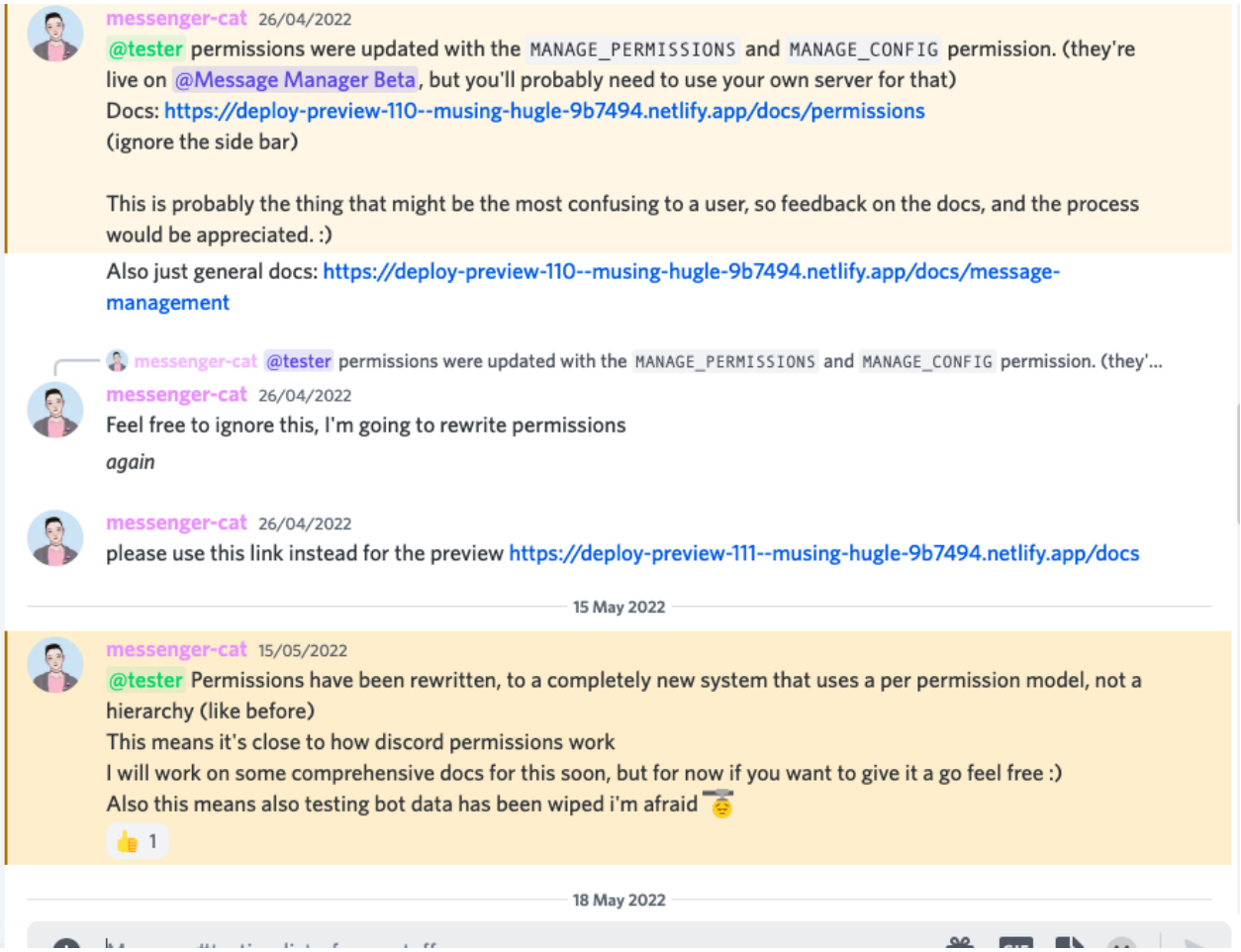


Figure 21 - Asking for feedback

In Figure 20 I'm asking for feedback from the users in my support server who have put their hands up for testing beta version of the bot. The feedback I did get was over voice call, so I do not have screenshots, but one significant aspect was the visual aspect of the permissions embed (see fig 19), before it was on one line and the feedback from that was it wasn't readable enough. Also, the second line that says that permissions have already been edited, but you can keep editing them with the same embed and select menus, was added after feedback so that it's clear the command doesn't have to be re-run each time.

Development Wrap-up

After carrying out extensive testing and implementing user feedback from testers (this feedback response is in each feature's section), I released the new version to users.

Throughout this process I was very careful, and provided the users with significant warning, and then when the migration and switch happened deployed a system the responded to old commands with instructions on how to use the new system.

Privacy, Security, and other Implications

Something that I kept in mind throughout the process was privacy and the impact my product could have on users. I addressed Security in the code level (see development section), and by ensuring that the platform I'm deploying on is secure, that I update it often, and that correct encryption processes are followed.

For Privacy I developed an extensive privacy policy, covering all data collected by the product, which also helped my review what data I'm collecting. I also make sure that all data isn't used for anything other than the purposes outlined in the privacy policy.

However, there is one area where this is significantly more difficult, and that is user generated content. As my product repeats user input, there is a high chance that someone is going to use that for malicious purposes, which could be hate speech, illegal content, etc. The issue with this is that it has the potential to cause harm to others.
To address this, I have initially directed users to come to me if they come across harmful content, but eventually I plan to implement a pre-emptive scanning system.

Message Manager – Scholarship Report
Summary and Reflection

Overall, I view this product and process as a success, as it has delivered an outcome that close to 1,000 different guilds are using it. It solves an issue that initially bothered me, and then caused me to create this product, which evidentially many others face. However, there are areas that can still be improved upon.

Comparison to Final Outcome Specification

- *Allow Staff Users to send, edit and delete messages*
  This has been achieved to its full extent.
- *Allow Staff Users to have some control over who can do the above using a permissions system*
  The permissions system currently provides an easy way in, but a lot of customisability for the advanced user. So yes, achieved fully.
- *Provide an intuitive user interface and flow – especially for new users*
  The user flow is much easier to use than the MVP, however there are still areas it's lacking in – especially in discord's area.
- *Documentation*
  Yes – all user facing functions are documented
- *Require the least technical know-how from a user as is possible*
  Yes – doesn't require coding know-how anymore

Comparison to Initial Task

"Create a service that allows a set group of users in a server to carry out message actions on a message from a shared account (sending, editing and deleting). The service will ensure that access to these actions can be customised by server staff, and the service must have a high-quality user interface and experience. The service must be able to be used by any server on discord."

The product has achieved most of these outcomes, but as above it is lacking in the "high-quality user interface and experience" to some extent. It is successful, and meets the task description, but has room to improve.

Next Steps / Changes

While this aspect of the project is over, there are still users who keep on using it, and the user base is growing, therefore I will continue to develop and improve this product in the future.

The biggest issue right now is the quality of the user interface, and while the current UI is very good, it could be a lot better. For example, the modal field for content input is restricted to only four lines, although it accepts up to 2000 characters, which is a limitation imposed by discord. Also, the embed editing process, while a leap forwards from the MVP, still requires quite a few clicks and interactions, which cost time, to

completed. Both these issues of UI are limited by Discord restrictions or features and are out of my control, to address this in the future as mentioned before I plan to build a web interface. This would allow me to have complete control over the UI, and design it to the products needs.

Reflection

The process I have followed throughout was effective and assisted in creating a high-quality product. I have improved in terms of my knowledge about different types of systems and programming techniques. I followed an incremental process, including the MVP stage, which allowed me to collect feedback and reflect on what I've done after each step, leading to a better more informed outcome. However this process resulted in a lot of duplicated work, or work that was eventually redone, which also reflects in the amount of time I've spent coding on this project. The start of this project, while it did have an overarching goal, lacked a focus on the end user, which was eventually rectified in the final product, also adding to the overall length of the project.

Conclusion

Considering the response from users who use the service, they have found that the bot has helped the to manage their servers well. That was my main goal with this project, to make something that others would find useful, something that solves an issue, and based off the positive feedback I have received, I managed to do that. I also have gained a lot of experience, something which is also important. The product is running well, and I'm continuing to maintain it and improve it, and I intend to keep doing so for a long time in the future.

Figure 22 - Rate of commands used over time - per day

This graph shows an increase in usage since the deployment of the final outcome.

References

[1] Discord, "Terms of Service," [Online]. Available: https://www.discord.com/terms.

[2] Discord, “Community Guidelines,” [Online]. Available: https://www.discord.com/guidelines.

[3] Discord, “Developer Policy,” [Online]. Available: https://discord.com/developers/docs/policies-and-agreements/developer-policy.

[4] Discord, “Developer Terms Of Service,” [Online]. Available: https://discord.com/developers/docs/policies-and-agreements/developer-terms-of-service.

[5] discord.py, “discord.py docs,” [Online]. Available: https://discordpy.readthedocs.io/en/v1.7.0/.

[6] discord.py, “cogs,” [Online]. Available: https://discordpy.readthedocs.io/en/v1.7.0/ext/commands/cogs.html.

[7] discord.py, “commands,” [Online]. Available: https://discordpy.readthedocs.io/en/v1.7.0/ext/commands/commands.html.

[8] Discord.JS, “Discord.JS,” [Online]. Available: https://discord.js.org/#/docs/discord.js/v12/general/welcome.

[9] Discord, “Slash Commands FAQ,” [Online]. Available: https://support.discord.com/hc/en-us/articles/1500000368501-Slash-Commands-FAQ.

[10] Cloudflare, “Tunnel,” [Online]. Available: https://www.cloudflare.com/products/tunnel/.