

Introduction

Our original goal was to build a fully functional chat application with authentication functionality, where channels were only accessible by authorized users. We also wanted to have some degree of channel and message modifiability, including channel deletion, message editing, and message deletion. To that end, we were able to achieve all of our goals, with users being able to create and delete their own channels, add other users to their channels, send real-time messages to all users in a channel, and edit and delete those messages in real-time.

Design/Implementation

We created a server in Go that hosts both the frontend and backend. For this purpose we used the web server from Go's native HTTP package. The frontend is served from the **public** directory on the filesystem, while the backend has several endpoints under the **/api/** route, as well as a websocket endpoint under **/ws/{token}**.

```
http.Handle("/", http.FileServer(http.Dir("./public/auth/")))
http.Handle("/chat/", http.FileServer(http.Dir("./public/")))
http.HandleFunc("/api/login", login)
http.HandleFunc("/api/register", register)
http.HandleFunc("/api/getData", getDataHandler)
http.HandleFunc("/api/getChannel/{channelID}", getChannelDataHandler)
http.HandleFunc("/ws/{token}", handleWs)
```

For data storage, we opted to use an SQLite database, as this seemed the simplest. You can see our database schema:

Name	Type	Schema
channels		CREATE TABLE channels ("uuid" TEXT NOT NULL PRIMARY KEY, "owner" TEXT NOT NULL, "name" TEXT NOT NULL)
uuid	TEXT	"uuid" TEXT NOT NULL
owner	TEXT	"owner" TEXT NOT NULL
name	TEXT	"name" TEXT NOT NULL
membership		CREATE TABLE membership ("channelUuid" TEXT NOT NULL, "userUuid" TEXT NOT NULL, UNIQUE("channelUuid", "userUuid"))
channelUuid	TEXT	"channelUuid" TEXT NOT NULL
userUuid	TEXT	"userUuid" TEXT NOT NULL
messages		CREATE TABLE messages ("msgUuid" TEXT NOT NULL PRIMARY KEY, "timestamp" DATETIME NOT NULL, "channelUuid" TEXT NOT NULL, "userUuid" TEXT NOT NULL, "msg" TEXT NOT NULL)
msgUuid	TEXT	"msgUuid" TEXT NOT NULL
timestamp	DATETIME	"timestamp" DATETIME NOT NULL
channelUuid	TEXT	"channelUuid" TEXT NOT NULL
userUuid	TEXT	"userUuid" TEXT NOT NULL
msg	TEXT	"msg" TEXT NOT NULL
users		CREATE TABLE users ("uuid" TEXT NOT NULL PRIMARY KEY, "email" TEXT NOT NULL UNIQUE, "password" BLOB NOT NULL, "username" TEXT NOT NULL UNIQUE, "salt" BLOB NOT NULL)
uuid	TEXT	"uuid" TEXT NOT NULL
email	TEXT	"email" TEXT NOT NULL UNIQUE
password	BLOB	"password" BLOB NOT NULL
username	TEXT	"username" TEXT NOT NULL UNIQUE
salt	BLOB	"salt" BLOB NOT NULL

Channel, message and user IDs are UUIDs generated using Google's UUID package. Usernames and emails are unique.

Once you log in or register, the **/api/login** or **/api/register** endpoint is called and the server issues you a session token. On registration, a new user entry is added to the users table. Passwords are not stored in plain text - they are hashed with Argon2id and salted with a random 16 byte salt. Session tokens are random 64-byte values stored as keys in a map where the values are user IDs, so they reset on server restart. Users authenticate themselves to the API via the sessionToken cookie. We opted to have the server check the identity of the caller instead of letting them supply their user ID, for security reasons. The WebSocket endpoint is called by supplying the token in the URL.

On initial app load, the client connects to the WebSocket endpoint at **/ws/{token}** to be able to get dynamic updates about message edits, deletions, user adds and channel subscriptions/creations, and then **/api/getData** is called to get data that is relevant to the current user - channels they're part of, messages in those channels, as well as what users are present in each of those channels.

Data	Length
↑ ("type":"message","arg1":"0cd566e3-1415-42e5-bf77-55bcc10a093e","arg2":"really cool ...	94
↓ ("action":"message","arg1":"79aff905-d2ab-4edf-b5d8-85add215db81","arg2":"user","arg...	190
↑ ("type":"message","arg1":"0cd566e3-1415-42e5-bf77-55bcc10a093e","arg2":"(dasfasdf)"	83
↓ ("action":"message","arg1":"3d5d952b-c528-406f-831c-a4ba5306320a","arg2":"user","a...	179
▼ {type: "message", arg1: "0cd566e3-1415-42e5-bf77-55bcc10a093e", arg2: "really cool message!"} arg1: "0cd566e3-1415-42e5-bf77-55bcc10a093e" arg2: "really cool message!" type: "message"	

Data	Length
↑ ("type":"message","arg1":"0cd566e3-1415-42e5-bf77-55bcc10a093e","arg2":"really cool ...	94
↓ ("action":"message","arg1":"79aff905-d2ab-4edf-b5d8-85add215db81","arg2":"user","arg...	190
↑ ("type":"message","arg1":"0cd566e3-1415-42e5-bf77-55bcc10a093e","arg2":"(dasfasdf)"	83
↓ ("action":"message","arg1":"3d5d952b-c528-406f-831c-a4ba5306320a","arg2":"user","a...	179
▼ {action: "message", arg1: "79aff905-d2ab-4edf-b5d8-85add215db81", arg2: "user",-> action: "message" arg1: "79aff905-d2ab-4edf-b5d8-85add215db81" arg2: "user" arg3: "1734400332215538000" arg4: "really cool message!" channelId: "0cd566e3-1415-42e5-bf77-55bcc10a093e"	

The valid actions for a user are: **message, edit, delete, channelSub, channelAdd, channelDelete.**

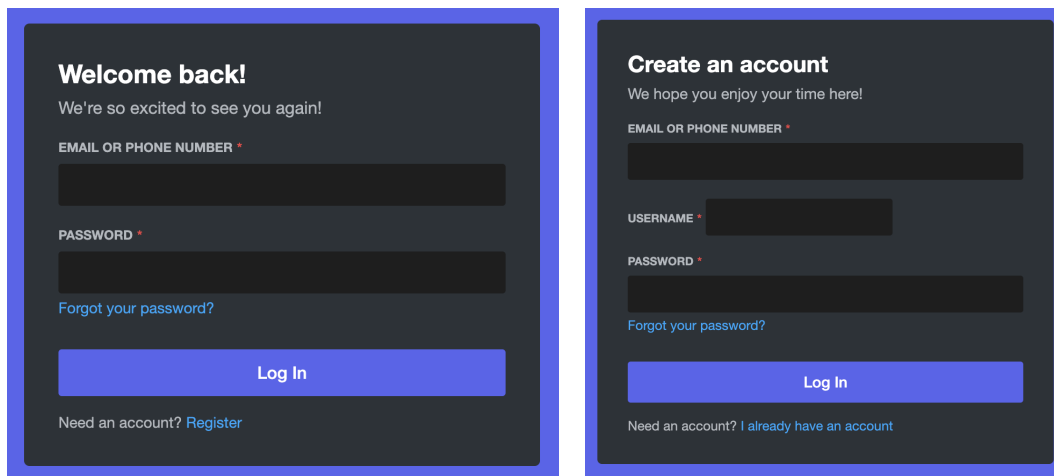
The valid server announcements are: **message, edit, delete, subscribe, createChannel, deleteChannel.**

If the user supplies an invalid authentication token, the API returns a 401 Unauthorized response. The server also performs input validation - for example, it will not allow a user to register if their password is shorter than 8 characters or if their email address is invalid. For actions regarding messages and subscribing another person to a channel, the server checks whether the user is a member of the channel they're trying to send a message or add another person to. Channel deletion is only allowed if the user is the owner of that channel.

The frontend is written in pure HTML, CSS, and JavaScript, and we made extensive use of event handlers.

Discussion/Results

Our final product is a minimal Discord clone which supports login/register operations, sending/editing/deleting messages, creating/deleting channels, and adding other users to your channels. When you first enter the page, you'll be asked to login, but if you don't have an account, then you can easily switch to the register page from the click of a button, which then shows three fields: email, username, and password. If the email and username have not already been used, and the password is of appropriate length, then you're simultaneously registered as a new user and logged in.



The image displays two side-by-side web forms for a Discord clone, both featuring a dark gray background with blue borders and accents.

Left Form: Welcome back!

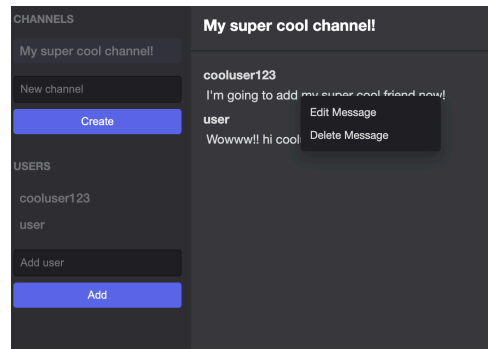
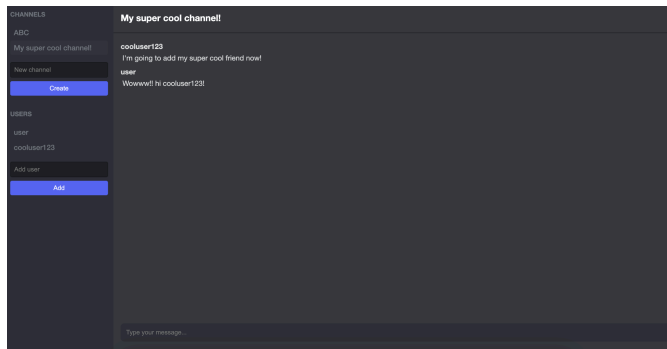
- Header: **Welcome back!**
- Text: "We're so excited to see you again!"
- Input field: "EMAIL OR PHONE NUMBER *"
- Input field: "PASSWORD *"
- Text: "Forgot your password?"
- Button: "Log In"
- Text: "Need an account? [Register](#)"

Right Form: Create an account

- Header: **Create an account**
- Text: "We hope you enjoy your time here!"
- Input field: "EMAIL OR PHONE NUMBER *"
- Input field: "USERNAME *"
- Input field: "PASSWORD *"
- Text: "Forgot your password?"
- Button: "Log In"
- Text: "Need an account? [I already have an account](#)"

After that, you're redirected to the chat page, which you are able to interact with as long as your session remains active (otherwise you're logged out and asked to return to the login page). On the chat page, you'll find a sidebar which contains a list of the channels that you're subscribed to, and also a list of users for the current channel that you're interacting with. Right clicking a channel will allow you to delete the channel as long as it is not the current channel and you were the creator of the channel.

The rest of the page consists of the chat area and the input bar, where you are able to send new messages. By right clicking on an existing message that you sent, you are able to edit or delete that message, which updates the screens of all other users looking at the channel. When you edit a message, an input interface appears where the message was, where you're able to type in your changes.



Our biggest challenges were mostly in designing the system to be both efficient and secure, as we had to make sure that all data such as session tokens and passwords that we were sending to the client could not be used by a malicious actor to penetrate the system. Our solution to this was to create temporary session tokens for each user login, which is stored in the client's cookies and expires when the user logs in from anywhere else. One example of this was when we were implementing the websocket connection, we were not able to access the cookies (which we use for authentication), so we had to include the session token into the endpoint.

Conclusions/Future work

Overall, we learned a lot about networking and web servers in Go, such as running the server on a specific port and creating different API endpoints to serve different types of data such as web pages and JSON. We also learned different ways to make our authentication and server more secure, such as obscuring as much information as possible from the client and handling all the operations on the server side, using api endpoints to serve specific information to the client. Furthermore, through using a SQLite database, we learned how to prevent potential attacks to our database and also how to concurrently handle database requests.

The project was definitely a very fun experience, as we were able to apply knowledge from the class and build a functional product. Though Snowcast, IP, and TCP were all great projects, our final product was just a command line interface, so it was much more satisfying to create a tangible final product.

If we were to continue working on this project, we would expand functionality to include more of what Discord has to offer including servers, adding friends, private messages, etc. With the way that the server is currently built, these features are well within our reach.