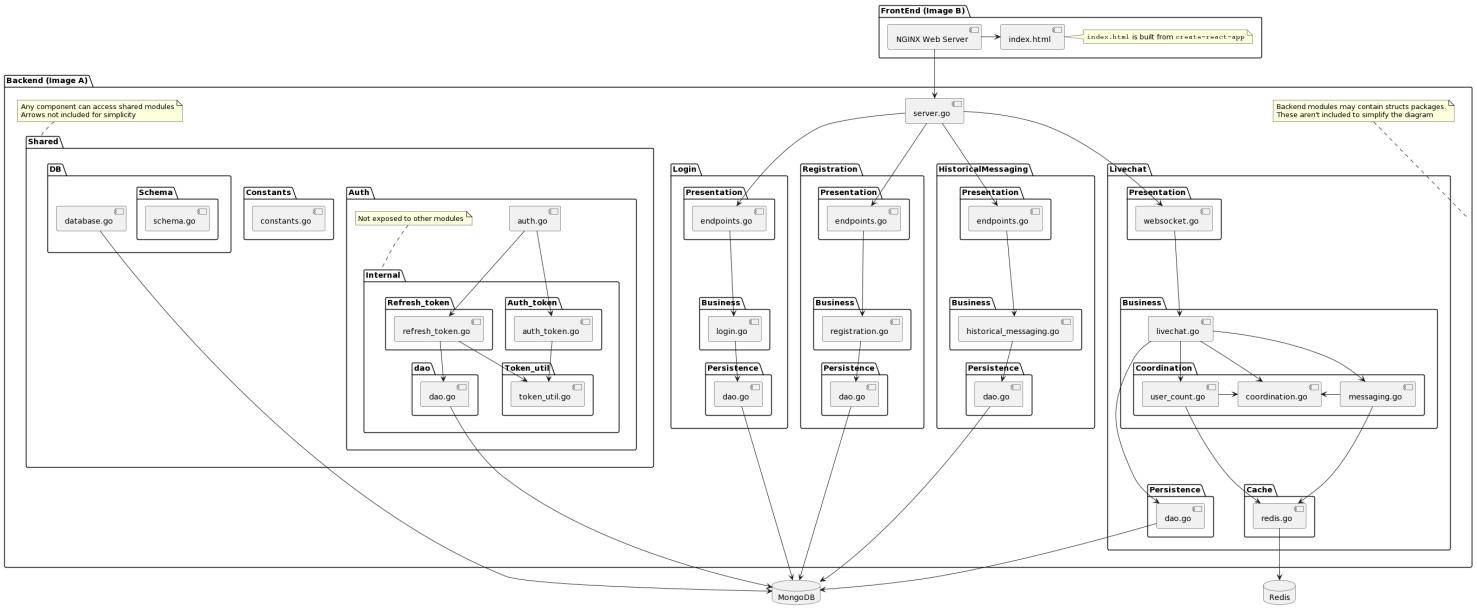
Distributed Systems Project 2 - Chat App

Name: Dante Laviolette | Student ID: 215717002

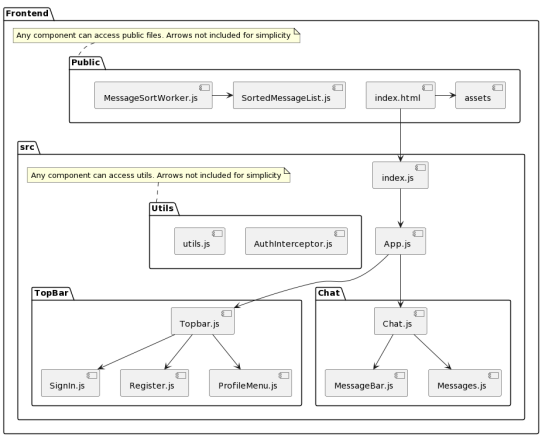
System Architecture

The system was designed using a layered architecture with limited coupling to allow for conversion to a micro-service architecture in the future. Each of the main backend modules contain a presentation, business, and data access layer. MongoDB is used for persistent storage, while Redis is used for caching & pub/sub communication between replicas . Due to the required coupling of certain services, and my following of the don't-repeat-yourself principle, I opted to create a few shared modules that any of the main modules can use.

The following is the entire systems architecture in the style of a simplified component diagram:



In the above diagram, the front-end is shown as `index.html`. This is because the front-end is written in React, and ultimately served in a single `index.html` file (which links to JavaScript files, and styles) built by `create-react-app` in the `Dockerfile`. The following is a component diagram of the front-end:



Component Design/Implementation

As authentication required functionality from other modules, this section will also explain the extra features design/implementation. This will only mention how the shared modules are used as they are more-so utilities than their own components. Although it's worth mentioning the `shared/DB` module returns the MongoDB client & collections, `shared/constants` returns constants, and `shared/auth` provides authentication functionality (producing/consuming credential tokens).

`Login`

- Provides the `POST /api/login`, `POST /api/logout`, & `GET /api/refresh_credentials` endpoints.
- The `/api/logout` endpoint calls `shared/auth` to validate the clients credentials, and then calls `shared/auth` to invalidate the users refresh token (delete it from the db).
- The `/api/login` endpoint takes the users details (email, name & password). It first validates the users password & email, gets the user with the given email from the DB, and checks if the given password matches the stored password hash. If so, it calls `shared/auth` to generate the users credentials for the response.
    - `shared/auth` creates an auth token returned in the `Authorization` header. This is a JWT (json web token) -- a JSON signed by a private key. This token is short-lived (15 minutes) and contains information about the user (`id`, `email` & `name`).
    - `shared/auth` generates a cryptographically random secret for the refresh token, storing its hash in MongoDB.
    - `shared/auth` creates a 1-time use refresh token returned as an `HttpOnly` cookie. This is a long-lived (30 day) JWT containing the `userId`, `secret` (plaintext secret) & `secretId` -- the id of the secret stored in MongoDB.
    - > Note: With this scheme, `shared/auth` provides middleware for validating these tokens, and re-generating the auth & refresh token upon the refresh tokens expiry. To re-generate (refresh) the credentials, it uses `FindOneAndDelete` to find the `secretId` in MongoDB, and refreshes the credentials if the refresh tokens `secret` matches the stored secret hash.
- The `/api/refresh_credentials` simply calls the `shared/auth` middleware that validates the clients credentials. The `shared/auth` middleware will automatically refresh the credentials if needed.

`Registration`

- Provides the `POST /api/change_password` and `POST /api/register` endpoints.
- The register endpoint validates & hashes the users password, and attempts to create a user for them in MongoDB. This fails if the email already exists and otherwise returns a 200 status code.
- The change password endpoint calls `shared/auth` to validate the clients credentials and then validates & hashes the new password, updating it in MongoDB based on the clients credentials.

`LiveChat`

- Provides the `/ws/chat` WebSocket. On initialization, it subscribes to the `messaging` & `userCount` Redis channels. When it receives a message from these channels, it broadcasts the message or user count to all of it's current sockets.
- The user count works by storing local variables to keep track of the authenticated users (`map`) and anonymous users (`int64`) on the current server. On (dis)connection/authentication, it updates the local values accordingly, (de)increments a global Redis value and then publishes the global updated user counts to the `userCount` Redis channel -- to be broadcasted to the sockets of all the replicas. The reason for authenticated users being stored in a map, is that the authenticated user count should only change when all sessions are ended for a user, or the first session begins.
    - On exit (ie. `SIGTERM`), it closes all connections and cleans up these values from the global Redis values to ensure the user counts stays accurate.
- On a new connection, it adds the socket to a local variable and deletes it on disconnection. This way, it can message all of its sockets when needed.
- On `ping` messages, it responds with a `pong` to keep the connection alive.
- On `auth` messages, it calls the `shared/auth` module to authenticate the user and then stores the authentication info (`authCtx`) for the socket. If successful, it sends an `auth` message. If the credentials are stale, it sends a `refresh` message.
- On `message` messages, it verifies that the `authCtx` was set (ie. the user is signed in), gets the current nanosecond timestamp, and uses that context along with the message to create a message from that user. It then publishes it to the `messaging` channel so all replica pods (including itself) will broadcast the message to their sockets, and finally it saves the message to MongoDB. It sends `message-failed` to the socket if anything fails.

`HistoricalMessaging`

- Provides the `GET /api/messages?lastTimestamp` endpoint. As `LiveChat` writes all messages to MongoDB, this endpoint takes a timestamp, finds the 50 messages that were sent before the `lastTimestamp` from MongoDB and returns them.

`Frontend/Chat`

- The chat page automatically calls `GET /api/messages?lastTimestamp=${now}` to load a page of historical messages, and starts a WebSocket connection to `/ws/chat`. It then sends a `ping` to the WebSocket every 500ms to keep the connection alive, and retries it's connection if it's lost.
- If the user is logged in, an `auth` message will be sent on the WebSocket to the server, so the `LiveChat` can authenticate them. If the client WebSocket receives an `auth` message, they are authenticated and the message bar UI will be enabled. If the client WebSocket receives a `refresh` message, they will refresh their credentials via `GET /api/refresh_credentials` and send an `auth`

message. If they don't receive a message, they are still considered a guest.

- When the client socket receives a message, it will be inserted to the `messages` linked list (sorted by its timestamp in a service worker) and then displayed in the UI.
- If the user scrolls to the top of the chat page, it will make a request to `GET /api/messages?lastTimestamp=${messages[0].ts}` to load another page of messages, and add them to the `messages` array, displaying them in the UI.
- When the user sends a message, a `message` is sent to the WebSocket with the subject & message.
- If the client WebSocket receives a `message_failed` message, an error will be displayed notifying them that their message failed to send.
- If the client WebSocket receives a `user_count` message, it updates the `AuthorizedUser` & `AnonymousUser` count in the UI.

`Frontend/TopBar`

- If user credentials are located in `localStorage.auth`, the `ProfileMenu` will be displayed allowing them to change their password or logout via API calls to `POST /api/change_password` and `POST /api/logout`.
  - Note that the `AuthInterceptor` will automatically set the requests `Authorization` header based on the `localStorage.auth` value.
- If the user isn't logged in, they'll see a login & register button providing functionality through API calls to `POST /api/login` and `POST /api/register`.
  - The `AuthInterceptor` will automatically store the returned auth token in `localStorage.auth` on success.

Component Interaction

The main backend modules do not interact with one-another aside from the usage of the shared modules. The only exception to this is the `server.go` entry point which defines the endpoints/WebSockets for each of the main components. Furthermore, the `LiveChat` communicates with it's own replicas using Redis Pub/Sub.

Looking at the entire system, all of the main backend modules interact with the MongoDB pod, writing or reading data, and only the `LiveChat` module interacts with the Redis pod. Looking at the front-end, it makes API/WebSocket calls to the NGINX server which acts as a reverse-proxy to `server.go` on `/api/*` or `/ws/*` endpoints, and otherwise serves static files, defaulting to `index.html`.

Performance Optimizations

- Historical messages are only loaded as the user scrolls up the chat page, and they are queried using paging to ensure only a limited amount of data is sent in each request.
- On the front-end, messages are stored in a custom doubly `LinkedList` data structure (`SortedMessageList`) which provides sorted insertion methods. As generally (ie. in non-race conditions) only the newest or oldest message is received, there is an insertion method for appending an assumed old message, as well as an assumed new message. These are worst-case O(n) methods, but O(1) when race-conditions don't exist.
    - The sorting logic mentioned above occurs in a service worker to avoid blocking the UI.
    - When the sorted messages are received from the service worker, the `useTransition` react hook is used to update the message list. `useTransition` will render the updated state while displaying stale state if the user interrupts it. In other words, this renders new messages while making the user feel as though the UI isn't being blocked.
    - Messages are rendered using `memo` (memoization) in React to avoid re-rendered messages.
    - All of these optimizations were necessary to ensure the UI stays responsive during periods of high message throughput.
- When socket messages are received on the server, their logic is executed in a new go routine to avoid blocking the sockets thread.
- Each `LiveChat` socket has its own channel and go routine for sending messages. This ensures consistency of the order of messages to that user while improving performance. When a message is sent to all sockets, the messages are simply written to the channels, so the current thread doesn't have to wait for all messages to send and can quickly proceed.
    - This also allowed the user count & messaging Redis subscriptions to constantly check for new data in background go-routines, and then notify the sockets without worrying about race-conditions or multiple routines writing to the same socket.

Resources

- Docs: Golang, Fiber Library, JWT Library, Mongo Library, Redis Library, and JoyUI Library
- YAML Templates: MongoDB YAML Template & `chatapp.yaml`
- My own previous project (in recent memory): Frontend & Backend

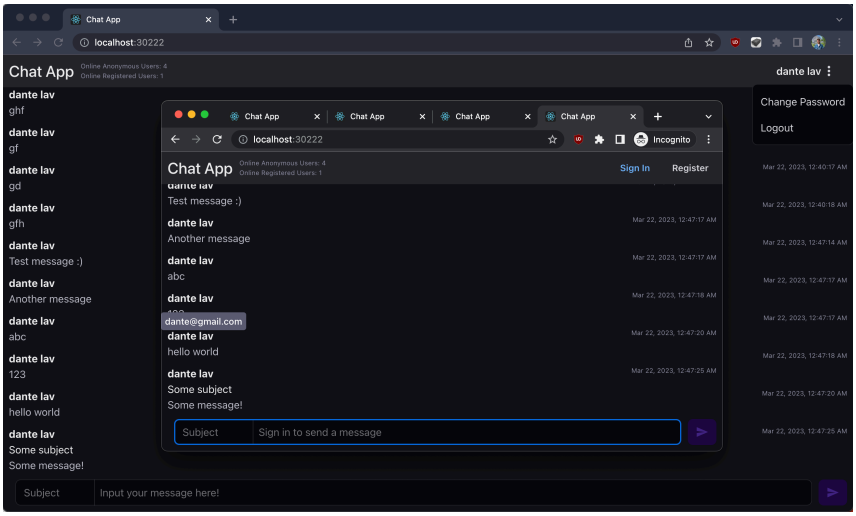
Testing Additional Feature: Authentication & User Count

The authentication feature can be tested by clicking the register or login button and following the screen prompts. Upon registering a new account, you'll be automatically signed in. You can then click the "..." in the menu bar to change your password or logout. As an aside, you can't send messages as an anonymous user, and can instantly begin sending messages upon signing in/registering.

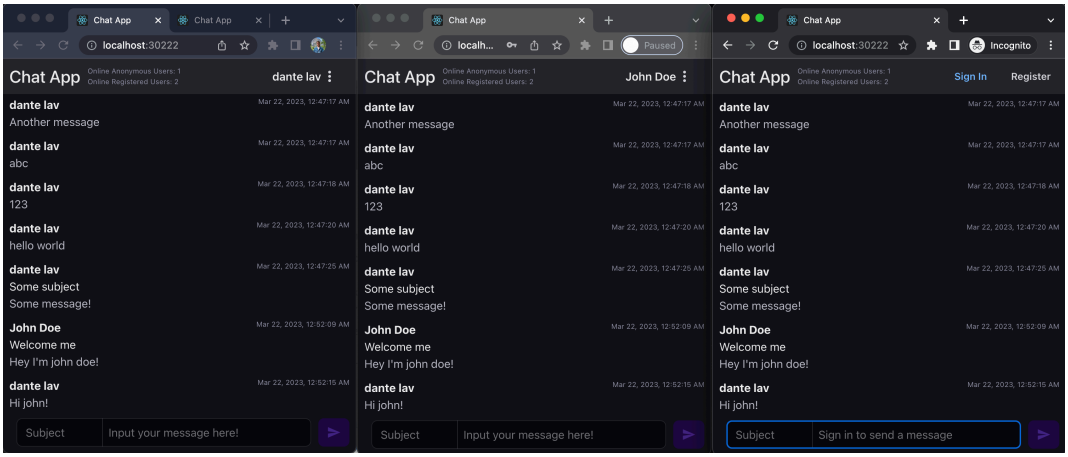
The user counts feature can be tested by opening/closing new browser windows (either signed in or signed out) and watching the counters change. It's important to note that the authenticated user-count is on a per-account basis -- so multiple sessions on the same account only count as one authenticated user (ie. the count only changes when all of that users sessions end, or their first session begins).

The following image shows the online user counts with 1 registered user and 4 anonymous users -- it can also be seen that one of the users (dante lav) is signed in and able to send messages, while the anonymous users must "Sign in to send a message":

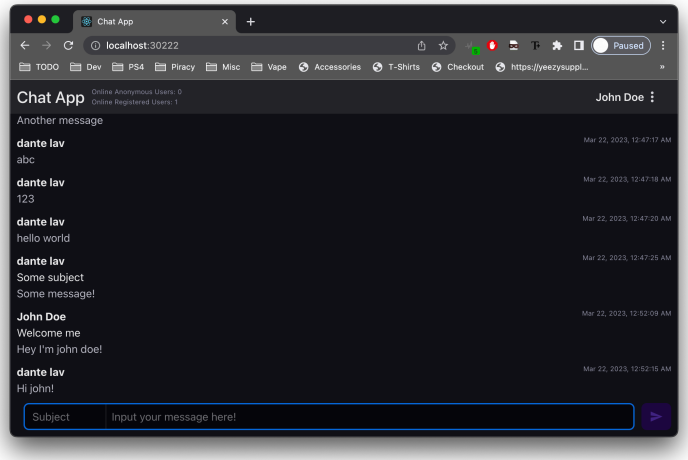
> Note: I made a small UI change since taking these screenshots. The email is now included next to the users name.



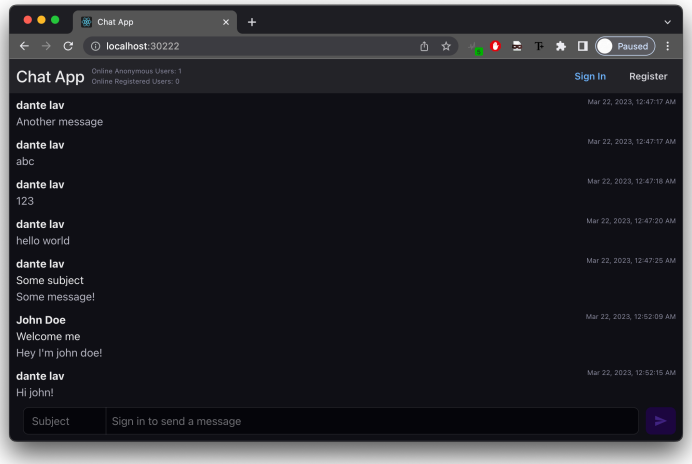
For this next image, I closed 3 of the anonymous user tabs, opened a new tab on Dante's account, and opened a new window with a new account -- as we can see there are now 2 registered users (the 2 tabs for Dante are only counted as 1 user), and 1 anonymous user. Furthermore, the same number is updated in real-time on all the open windows:



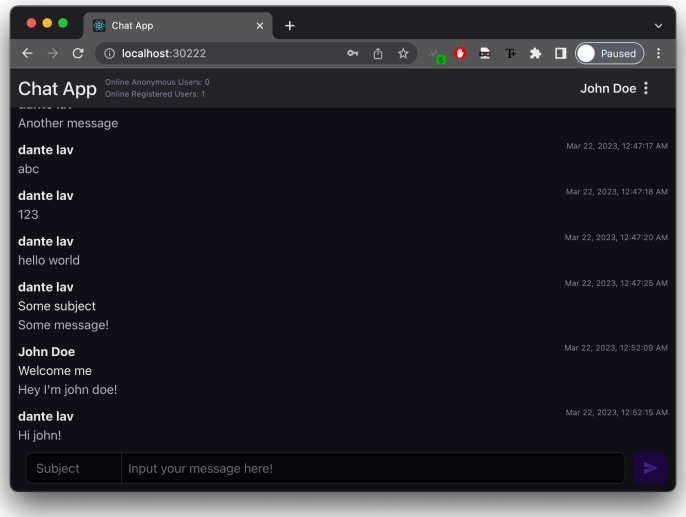
Next, I closed Dante's window (2 sessions for the same user) along with the window with the anonymous user. It can now be seen that only 1 registered user is now online:



I then logged out of Johns account, causing the count to update to 1 anonymous user and 0 registered users:

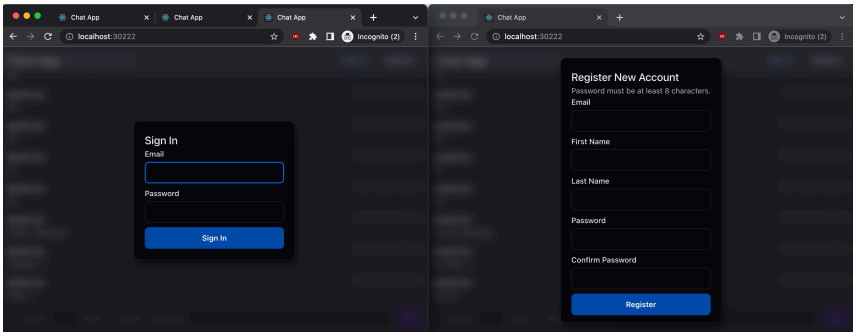


Finally, I logged back into Johns account, causing the count to update to 0 anonymous users and 1 registered user:

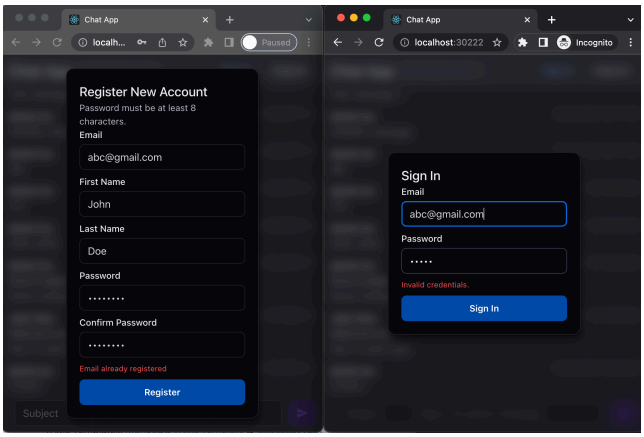


The following image shows the sign in & registration UI (upon clicking the "Sign In" or "Register" buttons):

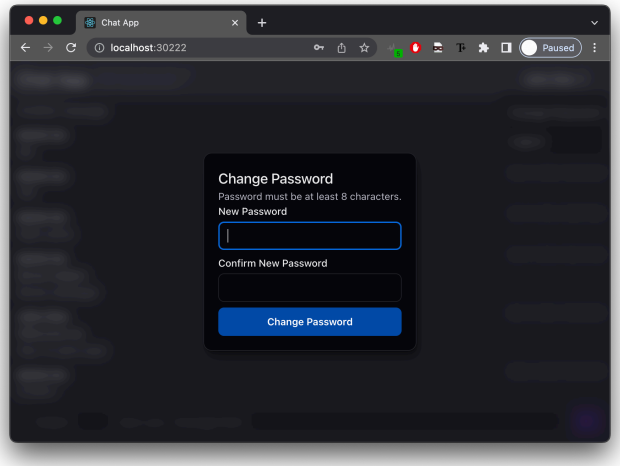
> Their functionality can be seen in the above images where I signed into Dantes account & registered a new account for John Doe.



Example of error handling when given bad input in registration/login UI:



The following image shows the change password UI (upon signing in, clicking "..." and then clicking "Change Password"). You can test this by logging out of the account after changing your password, and logging in again with the new password.


Chat App
localhost:30222
Paused
Change Password
Password must be at least 8 characters.
New Password
Confirm New Password
Change Password