Talk-To-Me Backend

Talk to me backend was developed with Flask version 1.1.2 and SocketIO version 2.3.0.
Here deals with the matchmaking and socket events for users to connect and send messages. For storing information, it uses MongoDB Atlas.

REST Endpoints

GET /auth

user_auth()

- Generates an auth for each user that joins goes to talk-to-me main page
- Using the user's IP address, it checks if the user is banned by checking the banned queue
- Adds newly created user into the database

Response Codes:

200: Success (user added into database)

POST /likes

handle_message_like(body)

- Makes sure the user exists
- Changes message into a liked state in message collection
- Emits 'message_liked' using socket.io

Response Codes:

200: Success (Message liked)
403: Forbidden (User needs to do /auth first)

POST /messages

handle_message(jsonObj)

- Makes sure the user sending a message exists
- Stores message in message collection and emits the 'send_message_to_client' using socket.io
- Checks if the user is banned - if they are, their messages will be sent to a bot

Response Codes:

201: Created (Message has been sent)
201: Created ((Banned user) Message has been sent to a bot)
403: Forbidden (User needs to do /auth first)

POST /queue

request_queue(body)

- Makes sure the user requesting to be queued exists in the database
- When user clicks on their preferred queue, they will be placed in that queue

Response Codes:

200: Success (User has been placed in queue)
200: Success (User has been banned)
403: Forbidden (User already in queue)
403: Forbidden (User needs to do /auth first)

POST /reports

handle_report(body)

- Determines which user in the database has sent a report
- Inserts the report information (reporter's info, reporter's reason, and reporter's room ID) into the report database
- Obtains the user's room ID and clones the conversation into the reported_messages database

Response Codes:

200: Success (User report has been sent)
404: Not Found (User who sent report cannot be found in the database)

Socket Events

"disconnect"

user_disconnect()

- If user disconnects/closes the tab while in queue, their information is deleted from the database
- If user disconnects/closes the tab while in a room, their information isn't deleted until the other user decides to disconnect

"hello"

user_sid_assoc(secret)

- Updates user's session ID by obtaining the user's assigned request ID

"join_room"

user_join_room(secret)

- Have the user with the provided secret join their room
- Emits socket event that the user has connected to a room

“leave_room”

user_leave_room(secret)

- Have the user with the provided secret leave their room
- Emits socket event that the user has disconnected from a room

Additional Methods

change_vent_listen_to_talk(query):

- Helper function to check_queue()
- Takes in a query and changes all matching documents’ queueType that have their wait time greater than 10 seconds to ‘talk’

check_queue()

- Deals with the matchmaking for Vent/Listen and Talk
- Continuously runs every 3 seconds using the APScheduler with id=check_is_queue_ready
- Users with ‘talk’ queue type gets matched with other users with ‘talk’ queue type
- Vent/Listen queue, tries to match with each other
    - If there is no match after a certain amount of time, will change their type to talk and get matched with talk

check_users_in_room(room_id)

- Ensures that users and room information are only deleted when both users have disconnected

delete_user_from_db(userObj)

- Deletes document with matching userObj from database

find_time_difference(userTime)

- Helper function to change_vent_listen_to_talk(query)
- Determines whether a user has been in queue for more than 10 seconds
- Returns true/false

is_banned(user_ip)

- Checks if the user’s IP address is in the banned database

match_making(userIDs)

- Changes two users (userIDs) queueType to outQueue
- Creates a document in room DB for these two users with their corresponding room

notify_queue_complete(user_id)

- Emits socket event that their queue is complete for both users in user_id

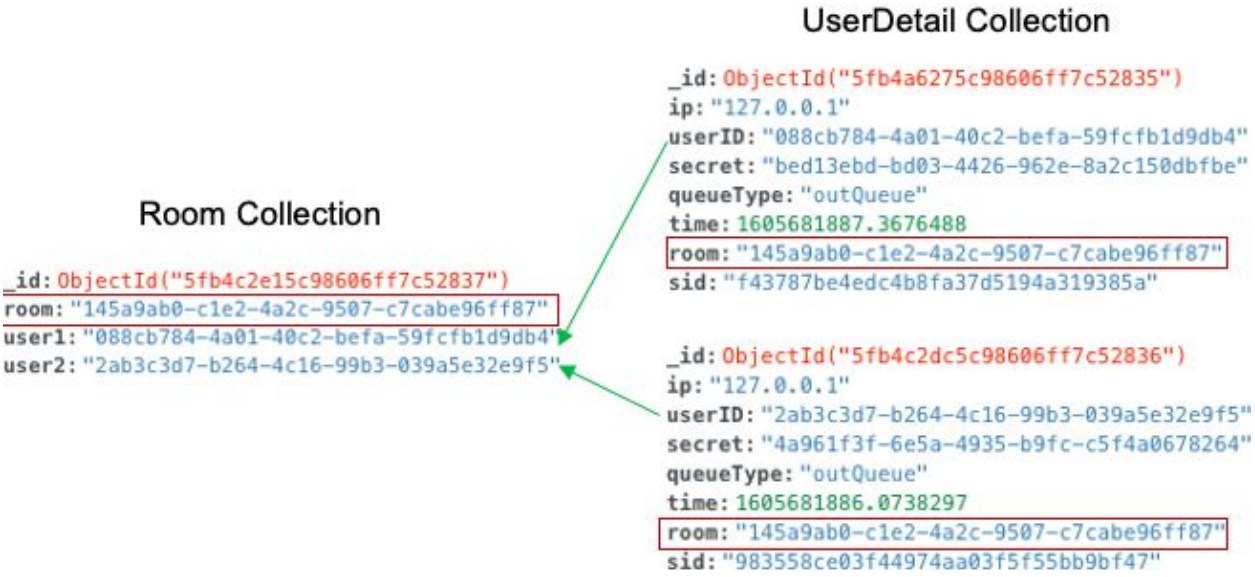
Matchmaking

When a user reaches the landing page, a user ID and user secret are automatically generated for them and stored in the database. The following is an example of document entry in the database:


```
_id: ObjectId("5fb4a6275c98606ff7c52835")
ip: "127.0.0.1"
userID: "088cb784-4a01-40c2-befa-59fcfb1d9db4"
secret: "bed13ebd-bd03-4426-962e-8a2c150dbfbe"
queueType: "idle"
time: 1605674535.6913917
room: "lonely"
sid: "f43787be4edc4b8fa37d5194a319385a"
```


Notice that the queueType field is set to “idle” by default, and the room field is set to “lonely”. This means that the user has only reached the landing page and has not been placed in the queue yet.

Once the user chooses either “Talk”, “Listen”, or “Be Heard” their queueType is updated accordingly. The database is queried every 3 seconds to find users with a defined queueType. Let’s say that two users have gone onto the website and clicked on “Talk”. The backend takes the two users and creates a room for them with a generated room ID. The users’ room fields are updated with the room ID and their queueType fields are updated to “outqueue”.



When the database is queried again, these users will not show up since they have already been placed in a room.

The “Listen” and “Be Heard” queues work a bit differently. Users in the “Listen” queue will be paired with users in the “Be Heard” queue (and vice versa). These queues are checked every 3 seconds as well, but an additional measure is put in place. If a user has been in the “Listen” or “Be Heard” queue for over 10 seconds with no success of being paired with someone else, their queueType field is set to “Talk”. This is put in place since the “Talk” queue will more than likely have the most people, while the “Listen” and/or “Be Heard” queue may be empty at times.

Therefore, every time the database is checked, the backend checks how many users have the queueType “Listen” and the queueType “Be Heard”. If both queues have at least 1 user in them, matchmaking can proceed as usual. However, if one queue does not contain any users, all users in the other queue will be checked to see if they’ve been in the queue for more than 10 seconds (using the users’ time field). All users that have been in the queue for more than 10 seconds will have their queueType updated to “Talk”.

Banning

When a user sends a report, the backend receives the report and stores it in the Reports collection. The report contains the following information:

- The reporter’s user ID
- The reporter’s IP Address
- The reported user’s user ID
- The reported user’s IP Address
- The report body/reason that the user submitted
- The room ID

Notice how the report contains the room ID. The backend proceeds to query the Messages collection using the room ID to find all the messages that were sent between both users in this room. The conversation is essentially cloned into the Reported Messages collection. This cloning is required since all messages are deleted after both users have left the room.

At this point, the backend’s work is complete, and it is up to the developer to determine whether or not to ban the reported user. The developer will read a report and query the Reported Messages using the room ID in order to read the chat conversation.

[Insert Visual]

If the developer believes that the reported user should be banned, they simply add the reported user's IP address, the reason for banning the user, and a date/time into the Banned Users collection. The /auth endpoint includes a ban check for every user who goes to the landing page. If the user's IP address is in the Banned Users collection, their queueType is automatically set to "banned".

{_id: automatic,
ip: "127.0.0.1",
reason: "blah",
date: "banned date"}

Before any user is placed in the queue, a ban check is performed. When a user with the "banned" queueType clicks on "Talk", "Listen", or "Be Heard", their queueTypes actually never get updated. Instead, they are matched with a bot and are free to message what they want; they are essentially shadow-banned.