Anonymous Chat Room

Christopher Allen

Michael Hamilton

University of Arkansas at Little Rock

Department of Computer Science

CPSC 3384-9S1 – Computer Networks

Dr. Francesco Cavarretta

April 15, 2025

Abstract

This report outlines the development of an anonymous chatroom application focused on real-time messaging, user safety, and accessibility. The system is built using Flask, Flask-SocketIO, Flask-Limiter, better_profanity, and React, with word suggestion support from the Datamuse API. These tools support an event-driven network architecture that manages persistent, two-way communication between clients and the server. Key data structures include bounded deques for storing recent chat history, Python dictionaries for tracking connected users and session data, and maps for assigning and storing user-specific color codes. These structures allow the server to maintain user state, synchronize messages efficiently, and provide a personalized experience for each client.

To support accessibility, especially for users with ADHD or attention challenges, the front-end features a color-coded message system, inline text formatting using custom tokens, and a toggleable light/dark mode. The interface design is guided by WCAG 2.1 accessibility standards and cognitive psychology research on readability and retention (W3C, 2023; National Library of Medicine, 2016). Safety features include a profanity filter, a message rate limiter, and secure file upload validation using MIME type checking and size restrictions. Together, these systems create a safe, anonymous, and inclusive chat environment. This project demonstrates how computer networking concepts can be used not only for functionality but also for building thoughtful, user-centered communication tools.

Introduction

This final phase of the Anonymous Chat Room project went beyond basic messaging and focused on combining networking technologies with real-time data handling, accessibility, and thoughtful interface design. Our main goal was to create a reliable and inclusive chat platform that supports a wide range of users, especially those with attention difficulties such as ADHD (NeuroLaunch, 2024), while promoting clarity, safety, and ease of use.

One of the core problems we wanted to address was cognitive accessibility in real-time digital communication. Many users, especially those with ADHD, struggle with information overload, poor message retention, and overstimulating interfaces. These challenges are not limited to one group; they affect a wide range of users, including those dealing with anxiety, screen fatigue, or neurodivergence (Sannon & Forte, 2022). Our design choices focused on reducing cognitive load, improving readability, and encouraging sustained engagement by using structured formatting, visual cues, and simplified layouts. The goal was to help users focus on key information, retain context, and feel safe and comfortable while participating.

From a technical perspective, we broke the system into sub-problems such as anonymous user identification, message synchronization, file sharing, profanity filtering, rate limiting, and front-end accessibility features. These were implemented using a stack of modern technologies including Python Flask for HTTP routes, Flask-SocketIO for real-time communication, Flask-Limiter for rate control, better_profanity for content filtering, and React for the client-side interface. Word suggestions were integrated using the Datamuse API, and accessibility features were informed by WCAG 2.1 and ADHD-focused research. Each component presented its own challenges in terms of performance, user experience, and behavior. We used purpose-built data structures and algorithms, and borrowed principles from psychology and accessibility research to guide our decisions at every level of development (He et al., 2020).

Problem Decomposition & Server Architecture

The chatroom backend was developed using Python's Flask framework, with real-time functionality handled by Flask-SocketIO, which enables persistent, bidirectional communication between the server and all connected clients (Grinberg, n.d.). Together, these components create a robust foundation for real-time messaging, file sharing, and anonymous interaction.

The server is responsible for handling multiple core features: managing user sessions, maintaining chat history, broadcasting messages, tracking online users, handling file uploads and downloads, and enforcing safety measures like profanity filtering and rate limiting. To accomplish this, we decomposed the server into distinct responsibilities, each backed by appropriate data structures and logic.

User Session Management

When a user connects to the chatroom, they are first given the choice to either enter a custom username or remain anonymous. This approach accommodates different types of users—those who wish to personalize their identity and those who prefer privacy or simplicity. If no name is provided, the system automatically generates a unique anonymous ID. If the user does not enter a name, the server calls a username generation function:

```
def gen_username():
    return "User-" + ''.join(random.choices(string.digits, k=4))
```

This function creates a generic but distinct username by appending a random four-digit number to the word "User." The resulting name ensures anonymity while still providing a unique identity that can be referenced during chat interactions. If the user enters a custom username, the server uses it instead of generating a random one. The logic that handles this decision:

```
@socketio.on('request_username')

def handle_custom_username(data):

    custom = data.get('custom', '').strip()

    username = custom if custom else gen_username()

    session['username'] = username
```

Once a username is confirmed (either custom or generated), it is stored on the server in memory using Python dictionaries:

- **`sid_username_dict`** maps each user's Socket.IO session ID (SID) to their associated username. This allows the server to track users by their unique session connection and ensures accurate identification for messaging, disconnection handling, and user-specific updates.

- **`user_colors`** assigns and stores a distinct visual identity for each user. Upon joining, each user is given a color from a predefined, accessibility-conscious color pool. This color is consistent across messages and the online user list, providing visual clarity and aiding in memory retention—especially useful in fast-paced or long chat sessions.

After a user is added, a system message is broadcast to all users announcing the new participant's arrival. This not only improves chatroom awareness but also builds a sense of real-time social presence.

```
join_message = {

    'username': 'System',

    'message': f"<span style='color: {user_colors[username]};
font-weight: bold;'>{username}</span> has joined the chat.",

    'user': username,
    'color': user_colors[username],
    'timestamp': datetime.now().strftime("%I:%M:%S %p")
}
chat_history.append(join_message)
socketio.emit('message', join_message)
```

This flow ensures that each user's entrance into the chat is:

- Personalized or anonymized based on preference
- Stored for session tracking
- Visually distinct via color-coding
- Immediately recognized by other participants through a broadcast

By combining session state management with real-time feedback, the application creates a smooth and welcoming experience for new users. When someone joins the chatroom, their session identity is stored, color-coded, and shared with other participants. This makes it easy to recognize new users and fosters a stronger sense of social presence from the start.

These session management features form the foundation of the chatroom's user identity system. They ensure every participant has a stable, recognizable presence, without needing accounts or passwords. This process also powers key features like message styling, visual consistency, and personalized interactions. This privacy-friendly design supports our broader goal of creating a clear, safe, and accessible communication space for all users.

Real-Time Messaging and Broadcasting

At the core of the chatroom is the `message` event, which enables real-time communication between all users. When a user sends a message from the front end, it is emitted to the backend through a persistent Socket.IO connection. The server listens for this event using the `@socketio.on('message')` decorator and begins a sequence of validation and packaging steps to prepare the message for broadcast. Each message contains a `username`: the sender's name,a `message`: the message content, a `timestamp`: when the message was sent, a `color`: the user's assigned visual color, and `validUsernames`: any mentions for highlighting. This allows the client to render the chat interface in a consistent, readable way, with support for color-coded usernames, timestamp alignment, and inline highlights.

To accomplish this, the first step is to ensure the incoming data is structured correctly. The server checks whether the received payload is a dictionary and contains a `"message"` key:

```
if isinstance(data, dict) and 'message' in data:
```

If the check fails, the message is ignored, and an error is logged to prevent invalid or malformed data from being processed:

```
else:
    print("Error: Received data is not a valid object or missing 
'message' field.")
```

If the message is valid, the server proceeds to extract key metadata:

- `username` is retrieved from the session.
- `color` is retrieved from the `user_colors` dictionary.
- `timestamp` is generated to reflect when the message was received.
- `validUsernames` is a list of detected `@mentions`, parsed from the message body.

The structured message is then assembled into a dictionary:

```
message_data = {
    'username': username,
    'message': message,
    'color': color,
    'timestamp': datetime.now().strftime("%I:%M:%S %p"),
    'validUsernames': mentions
}
```

Finally, the message is broadcast to all connected clients using Socket.IO:

```
socketio.emit('message', message_data)
```

This real-time delivery process ensures that all participants receive updates simultaneously, with each message displayed alongside consistent visual metadata such as username color and a timestamp. The use of a session-linked identity, combined with broadcasting, helps maintain a fast and organized messaging environment with minimal delay or confusion.

Message Handling Improvements

After implementing the core message flow, several enhancements were made to improve safety, clarity, and usability.

1. **Profanity Filtering**
   To promote a safe and respectful environment, all messages are scanned using the `better_profanity` library (Srivatsav, 2022). If a message contains offensive language, it is automatically censored before being shared with other users. This helps maintain a positive space.

2. **Message History**

   This ensures the server retains only the most recent messages, conserving memory and improving efficiency without requiring a database (Srivatsav, 2022). When new users join, they receive this recent message history (excluding system messages) so they can immediately see what others are talking about and join the conversation with context. The `message_data` object is added to the in-memory `chat_history` deque, which holds a limited number of recent messages for replay purposes when new users join:

   ```
   chat_history.append(message_data)
   ```

3. **Mention Parsing**

   Messages are scanned for `@username` mentions. Any detected mentions are included in a list called `validUsernames`, which the front end uses to highlight them. This social feature improves awareness and helps users direct attention to others without relying solely on message order (W3C, 2023).

Managing Online User Data

In addition to managing messages, the server keeps track of all currently connected users through a real-time online user list. This list is dynamically updated and includes each user's username and assigned color. The server generates this list using the in-memory `user_colors` dictionary, which stores the usernames and their corresponding color values. When users connect or disconnect, the list is reconstructed using a dictionary comprehension:

```
user_list = [{"username": u, "color": c} for u, c in
user_colors.items()]
socketio.emit('update_user_list', user_list)
```

This list is then emitted to all connected clients, ensuring every user sees an up-to-date view of who is online. The use of a dictionary makes this update process fast and efficient, even as users connect or disconnect frequently.

The online user list plays a vital role in supporting social presence and user awareness, allowing participants to see who else is currently active in the chatroom. This feature also enables functionality like direct mentions (`@username`) and ensures that each user's assigned color is consistently recognized across the interface. From a user experience perspective, the list helps users quickly identify who is present, making the environment feel more interactive and socially engaging. Because the system does not rely on external databases, these updates occur instantly as users join or leave, allowing the interface to remain lightweight, fast, and synchronized in real time.

File Uploads and Downloads

Uploads

Users can share files with others through the `/upload` endpoint. When a file is submitted via a POST request, the server performs several checks before accepting and broadcasting the upload. If a valid file is detected and passes all checks, it is stored with a unique name to avoid filename conflicts:

```
file = request.files['file']

filename = secure_filename(file.filename)

filename = f"{uuid.uuid4()}_{file.filename}"

filepath = os.path.join(UPLOAD_FOLDER, filename)

file.save(filepath)
```

After the file is saved, a media message is constructed and broadcast to all connected clients:

```
file_url = f"{request.host_url}download/{filename}"
media_message = {
    'username': username,
    'message': f"Shared a file: {file.filename}",
    'file_url': file_url,
    'timestamp': datetime.now().strftime("%I:%M:%S %p"),
    'color': user_colors.get(username, "#888")
}
chat_history.append(media_message)
socketio.emit('message', media_message)
```

This allows uploaded files to appear naturally in the chat stream alongside normal messages with the ability to download them.

Downloads

When a user clicks on a shared file link in the chat, the server delivers the file through the `/download/<filename>` route. This endpoint is specifically designed to safely handle file access requests and ensures that users can only retrieve files that were intentionally made available through uploads.

The route uses a secure helper function from Flask:

```
@app.route('/download/<filename>', methods=['GET'])
def download_file(filename):
    try:
        return send_from_directory(UPLOAD_FOLDER, filename)
    except Exception as e:
        print(f"[ERROR] Failed to serve file: {e}")
        return jsonify({"error": "File not found"}), 404
```

The `send_from_directory()` function ensures that files are only served from the designated `UPLOAD_FOLDER` and not from other areas of the file system. It helps prevent common web vulnerabilities such as directory traversal attacks (He et al., 2020) (trying to access `../../etc/passwd`) by isolating file access to a known, controlled directory.

By wrapping the logic in a `try/except` block, the server can handle missing or invalid file requests and return an error response. This approach reinforces security, improves reliability, and ensures that the server does not expose internal errors or file paths to end users.

Overall, this method keeps download logic clean, focused, and secure—allowing users to retrieve shared files without compromising the safety or structure of the backend.

File Safety & Security Enhancements

To ensure the chatroom remains safe and stable, several layers of protection are implemented around file uploads.

1. MIME Type and Extension Validation

The server verifies both the file extension and the MIME type before accepting a file:

```
ALLOWED_EXTENSIONS = {...} ie ‘jpg’ ‘png’ ‘mp4’ ‘pdf’ ‘zip’ ‘js’ and
much more

def allowed_file(filename):
    ext = filename.rsplit('.', 1)[1].lower() if '.' in filename else
''
```

```
    mime_type, _ = mimetypes.guess_type(filename)
    return ext in ALLOWED_EXTENSIONS and mime_type is not None
```

This prevents unsupported or malicious file types from being uploaded.

2. Filename Sanitization

To avoid injection attacks or directory traversal, all filenames are sanitized using:

```
filename = secure_filename(file.filename)
```

This function strips directory paths, removes unsafe characters, and ensures the result is a clean, safe, ASCII-only filename suitable for server storage. Additionally, a UUID is prepended to prevent name collisions:

```
filename = f"{uuid.uuid4()}_{file.filename}"
```

3. File Size Limit

Uploads are restricted to a maximum file size of 50MB to protect server memory and prevent abuse. This is enforced at the Flask config level:

```
app.config['MAX_CONTENT_LENGTH'] = 50 * 1024 * 1024
```

4. Rate Limiting

To prevent flooding, the `/upload` route is limited to **three uploads per minute per user** using Flask-Limiter:

```
@limiter.limit("3 per minute")
```

These security measures ensure that the chatroom remains safe, efficient, and stable.

Frontend Design & Real-Time Messaging

The client-side interface for the Anonymous Chat Room is developed using React, allowing for a modular, responsive, and highly interactive experience (Meta, n.d.). It maintains a persistent connection to the Flask-SocketIO server using a Socket.IO client. This enables continuous, real-time communication between users without the need for constant polling or page refreshes (Rauch, 2024).

Joining the Chatroom

When a user first accesses the chatroom, they are presented with a welcome screen that prompts them to either enter a **custom username** or continue anonymously. Once they proceed, the following call is triggered:

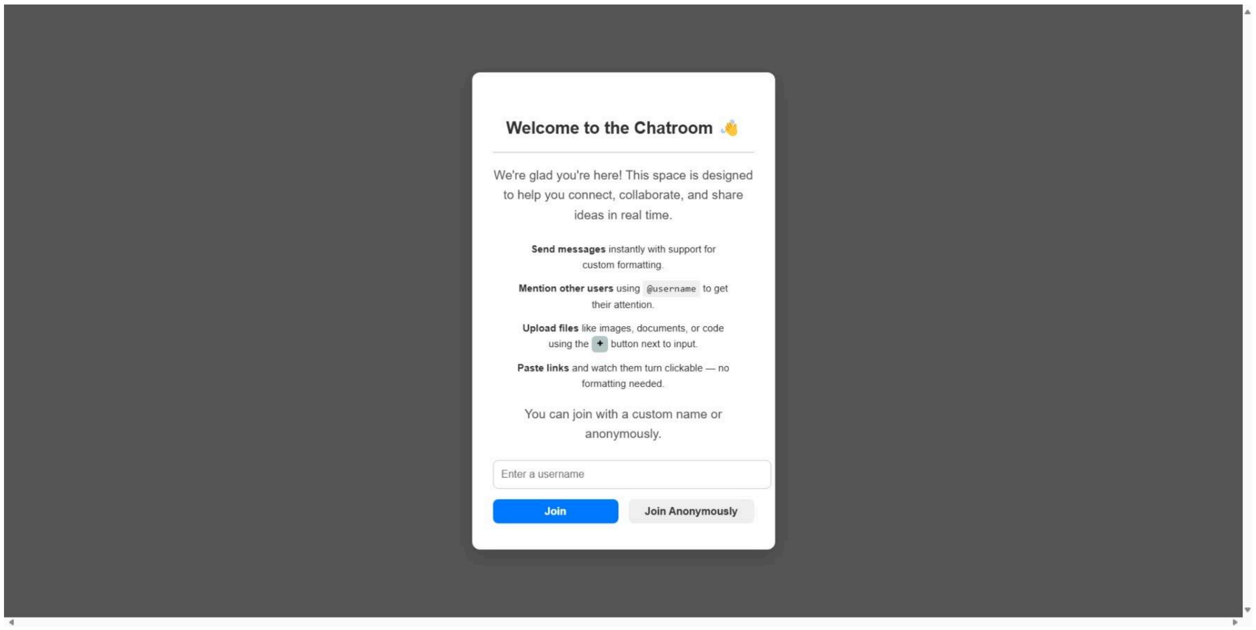
```
socket.emit("request_username", { custom: customName });
```

This sends their selected (or empty) username to the backend. The server responds with a final, validated username (randomly generated if none is provided) and emits a color from a rotating pool to help visually distinguish users (NeuroLaunch, 2024; National Library of Medicine, 2014). The client captures this with:

```
socket.on("set_username", (data) => {
    setUsername(data.username);
    userColors.current[data.username] = { lightColor: data.color };
});
```

This process ensures that every client is aware of their identity and can be visually distinguished by others, even in anonymous settings.

Figure 1. Welcome screen where users choose between a custom or anonymous identity.


Welcome to the Chatroom 👋

We're glad you're here! This space is designed to help you connect, collaborate, and share ideas in real time.

Send messages instantly with support for custom formatting.

Mention other users using `@username` to get their attention.

Upload files like images, documents, or code using the + button next to input.

Paste links and watch them turn clickable — no formatting needed.

You can join with a custom name or anonymously.

Sending a Message

When a user types a message and presses enter, the following function is invoked:

```
socket.emit("message", { username, message: input });
```

This packages the message and username into a payload and sends it to the server via the `message` event. Once the backend processes it), the same structured message is emitted back to all users. Each incoming message is handled through this listener:

```
socket.on("message", handleMessage);
```

This ensures real-time message synchronization across all clients.

The frontend dynamically formats messages using a custom `formatMessage()` function. This function highlights:

- `@mentions`
- Special tokens like `#highlight#`, `!warning!`, `$value$`, `~tone~`
- Filenames and URLs

Each of these is parsed and wrapped in a styled `<span>` using regular expressions. For instance:

```
messageContent = messageContent.replace(/#(.*?)#/g, (match, text) => {
    return `<span class="highlight-hashtag">${text}</span>`;
});
```

This modular formatting system gives users agency to **self-annotate** their content for clarity and priority, supporting users with focus difficulties or information overload.

Receiving the Online User List

In addition to messages, the server also emits updates to the user list. This update is interpreted on the front end by:

```
socket.on("update_user_list", (users) => {

    setOnlineUsers(users);

});
```

Each user is displayed in a sidebar with a persistent color associated with their username. This UI pattern reinforces social presence by helping users track participants in real time, even as people join or leave. These identifiers are important in fast-paced or anonymous environments, where distinguishing users can be difficult.

UI Design, Accessibility, and Cognitive Support

Beyond functional real-time features, the chat interface was carefully designed to support users with attention-related challenges (like ADHD) and visual impairments, adhering to WCAG 2.1 accessibility guidelines (W3C, 2023) and informed by cross-disciplinary research in psychology and HCI (Human-Computer Interaction).

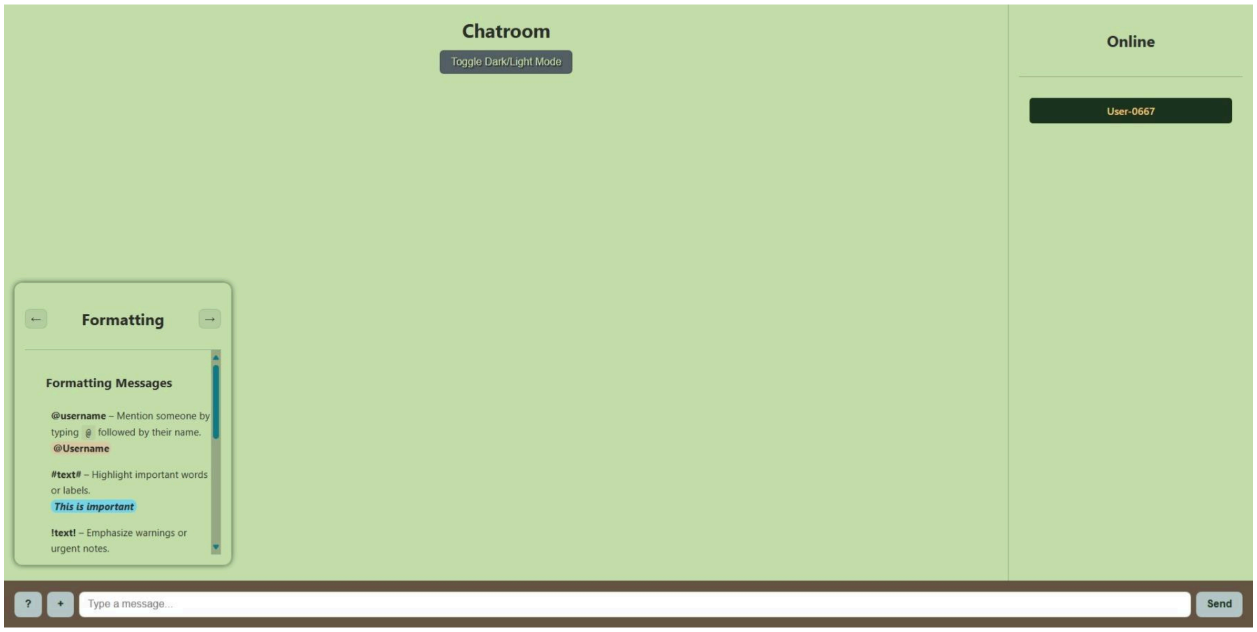
Light and Dark Mode

The interface supports a dual-mode color system that can be toggled at any time:

```
<button id="toggleMode" onClick={handleToggleMode}>
    Toggle Dark/Light Mode
</button>
```

These themes are built using CSS variables, ensuring consistent styling across components and minimal re-renders. All color schemes were tested using Adobe Color to ensure sufficient contrast and distinguishability for users with color vision deficiency (Adobe, n.d.

Figure 2. Light mode layout featuring WCAG-compliant color palette.



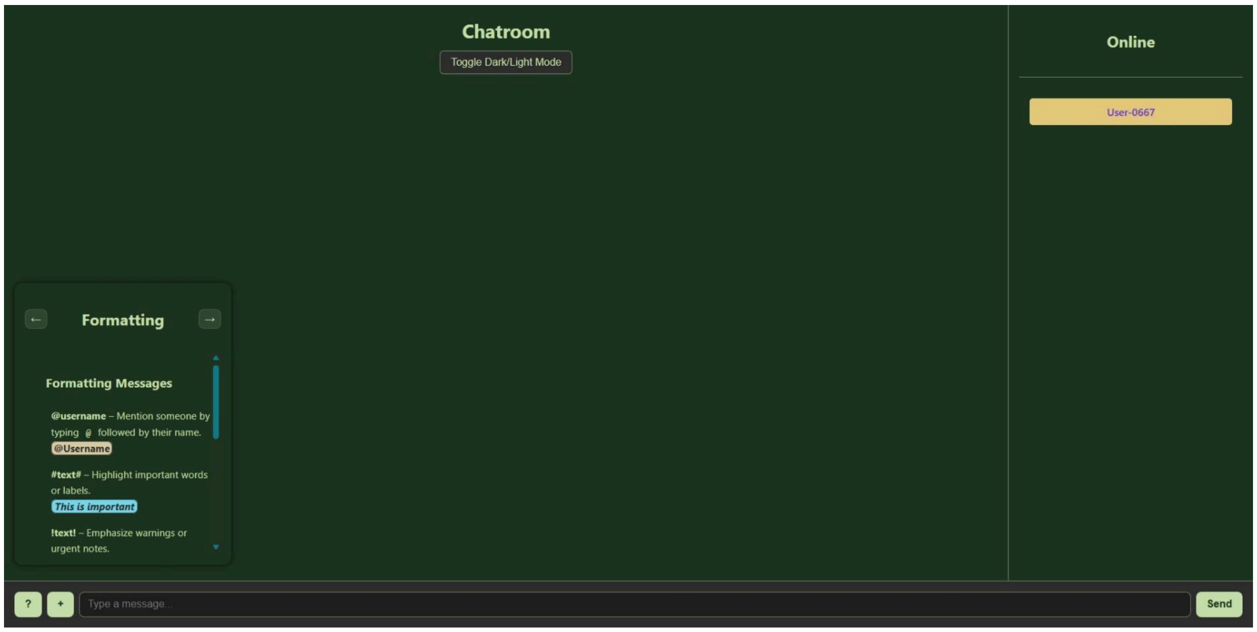
Formatting Messages

@username – Mention someone by typing `@` followed by their name.
@Username

#text# – Highlight important words or labels.
This is important

!text! – Emphasize warnings or urgent notes.

Figure 3. Dark mode layout optimized for low-light environments.



Formatting Messages

@username – Mention someone by typing @ followed by their name.
@Username

#text# – Highlight important words or labels.
This is important

!text! – Emphasize warnings or urgent notes.

ADHD-Informed Readability Features

We implemented several design decisions based on studies from *Frontiers in Human Neuroscience* and *NeuroLaunch* (NeuroLaunch, 2024; National Library of Medicine, 2014), including:

- **Message bubbles**: Each user's message appears in a speech-bubble style container on alternating sides (current user on the right), reducing visual clutter and increasing spatial clarity.

- **Consistent username color**: Users are assigned a persistent, readable color to improve tracking and recognition during conversations.

- **Fade-out system messages**: Temporary notifications (like user joins/leaves) automatically disappear after a few seconds to minimize noise.

Formatting via Token System

Inspired by markdown and ADHD-friendly formatting cues, users can emphasize content by wrapping text in lightweight tokens:

Token	**Purpose**	**Output Class**
#text#	Highlight/important	.highlight-hashtag
!text!	Urgent/warning	.highlight-exclamation
$text	Value or reward	.highlight-dollar
~text~	Alternate tone/sarcasm	.highlight-tilde
@user	Mentions	.highlight-mention

This system empowers users to control how their content is perceived, which is especially beneficial for neurodivergent users who may struggle to extract signals from noise in a chat interface (Behavioral and Brain Functions, 2014; NeuroLaunch, 2024).

Word Autocomplete (Language Support)

To assist users who experience memory lapses or delayed word recall, the client includes an autocomplete suggestion system powered by the Datamuse API (Datamuse, n.d.):

```
const res = await
fetch(`https://api.datamuse.com/words?sp=${prefix}*&max=10`);
```

Suggestions appear as the user types, and can be clicked to complete a thought. This small feature has an outsized benefit on expressive fluency, allowing users to participate more confidently and efficiently in fast-paced discussion.

Unified Architecture: Back-and-Forth Communication

The chatroom's architecture is built on an event-driven model that synchronizes the frontend and backend. This enables real-time interaction, reactive updates, and seamless coordination between users.

Here’s how this process works in harmony:

- **The backend emits stateful events**—such as user join/leave updates, chat messages, system notifications, and file URLs—using Socket.IO.

- **The frontend listens for these events**, immediately rendering them into the UI while applying accessibility rules (color contrast, message formatting, fade-out system messages).

- **User actions on the frontend**, sending a message or uploading a file, trigger client-side state changes. These are transmitted to the server, where they are validated (rate limits, profanity filtering) and rebroadcast to all clients.

- **All connected clients stay synchronized** through shared update events and message history, which is stored server-side in a bounded `deque` and sent to new users on connection.

This continuous loop of event emission and reception creates a responsive experience that adapts to user input in real time. It also supports robust session awareness, clear visual identity, and an inclusive user environment, delivering both performance and predictability across the application.

Conclusion

The Anonymous Chat Room is more than just a messaging platform, it is a networked system deliberately designed with psychological safety, usability, and real-time performance at its core. By breaking the problem into key subcomponents, selecting appropriate algorithms and data structures, and drawing insight from disciplines like cognitive psychology and accessibility research (Behavioral and Brain Functions, 2006; W3C, 2023), we built a robust and inclusive communication tool.

Its architecture demonstrates core networking concepts in action, message broadcasting, session handling, client-server state sync, and rate-limited event flows, while embodying values that prioritize the user. This project shows that effective computer networking design can extend beyond function into the realm of ethical, accessible, user centered systems.

References

Adobe. (n.d.). *Adobe Color*. https://color.adobe.com/

AudioEye. (n.d.). *AudioEye accessibility platform*. https://www.audioeye.com/

Behavioral and Brain Functions. (2006). *Impaired color perception in ADHD: Evidence from an object color-naming task*. https://behavioralandbrainfunctions.biomedcentral.com/articles/10.1186/1744-9081-2-4

Behavioral and Brain Functions. (2014). *Color perception and attentional load in ADHD*. https://behavioralandbrainfunctions.biomedcentral.com/articles/10.1186/1744-9081-10-39

Behavioral and Brain Functions. (2014). *Attentional dysfunction in ADHD and color sensitivity: A companion paper*. https://behavioralandbrainfunctions.biomedcentral.com/articles/10.1186/1744-9081-10-38

Datamuse. (n.d.). *Datamuse API: Autocomplete*. https://www.datamuse.com/api/autocomplete/

Grinberg, M. (n.d.). *Flask-SocketIO Documentation (Version 5.3.6)*. Flask-SocketIO. https://flask-socketio.readthedocs.io/en/latest/

He, Y., Zhang, M., Yang, X., Luo, J., & Chen, Y. (2020). A survey of privacy protection and network security in user on-demand anonymous communication. *IEEE Access, 8*, 54856–54871. https://doi.org/10.1109/ACCESS.2020.2981517

NeuroLaunch Editorial Team. (2024, August 4). *ADHD fonts: Enhancing readability and focus for individuals with Attention Deficit Hyperactivity Disorder*. NeuroLaunch. https://neurolaunch.com/adhd-font/

NeuroLaunch Editorial Team. (2024, August 4). *Best colors for ADHD: Color psychology and learning focus*. NeuroLaunch. https://neurolaunch.com/best-colors-for-adhd/

National Library of Medicine. (2016). *Visual perception in ADHD: A focus on color discrimination*. *Frontiers in Human Neuroscience*. https://pmc.ncbi.nlm.nih.gov/articles/PMC4947084/

National Library of Medicine. (2014). *Color perception in children with ADHD compared to neurotypical controls*. https://pmc.ncbi.nlm.nih.gov/articles/PMC3938738/

National Library of Medicine. (2014). *Visual processing in ADHD: Color contrast sensitivity*. https://pmc.ncbi.nlm.nih.gov/articles/PMC4282194/

Rauch, G. (2024). *Socket.IO (Version 4.8.1)* [Computer software]. Automattic. https://socket.io/

Meta. (n.d.). *Describing the UI*. React Documentation. https://react.dev/learn/describing-the-ui

Sannon, S., & Forte, A. (2022). Privacy research with marginalized groups: What we know, what's needed, and what's next. *Proceedings of the ACM on Human-Computer Interaction, 6*(CSCW2), Article 455. https://doi.org/10.1145/3555556

Srivatsav, D. (2022, February 19). *Simple chat room using Python*. GeeksforGeeks. https://www.geeksforgeeks.org/simple-chat-room-using-python/

W3C. (2023). *Web Content Accessibility Guidelines (WCAG) 2.1 Quick Reference*. https://www.w3.org/WAI/WCAG22/quickref/?versions=2.1