22AIE204 INTRODUCTION TO COMPUTER NETWORK

A PROJECT

***Submitted by Group 11***

***AIE BATCH A***

SYED ANEES ASHRAF (CB.SC.U4AIE23068)

C V SHREYAAS ADITYA (CB.SC.U4AIE23018)

B SAI SARAN GOUD (CB.SC.U4AIE23017)

S G SATHWIK (CB.SC.U4AIE23064)

*ON THE TOPIC*

# Building a Real-Time Chatting System

BACHELOR OF TECHNOLOGY IN

COMPUTER SCIENCE ENGINEERING – ARTIFICIAL INTELLIGENCE



**Department of Artificial Intelligence** AMRITA VISHWA VIDYAPEETHAM COIMBATORE - 641 112 (INDIA)

# ACKNOWLEDGEMENT

We would like to express our sincere appreciation to all those who contributed to the successful completion of this project. This endeavor would not have been possible without the collective efforts, dedication, and expertise of the following individuals

We extend our gratitude to each member of the project team for their hard work, collaboration, and commitment to excellence. Their diverse skills and perspectives greatly enriched the project and ensured its successful execution.

We would like to acknowledge Ms. B Ganga Gowri, Assistant Professor, CEN for her collaboration and contributions to specific aspects of the project. Her expertise and cooperation significantly enhanced the overall quality of our work.

We are grateful for the support received from Amrita Vishwa Vidyapeetham. The resources, facilities, and conducive environment provided by the institution were instrumental in the successful execution of our project.

To our friends and family, who stood by us with patience and encouragement, we extend our heartfelt appreciation. Your understanding and support were invaluable during the demanding phases of this project.

In conclusion, each individual and entity mentioned above played a crucial role in the realization of this project. Our gratitude extends to everyone who contributed to various capacities, and we are proud to have worked with such a dedicated and talented team.

**CONTENTS**

**Chapter Title Page No.**

1. Introduction 4
2. Methodology 6
3. Implementation 9
4. Results and Discussions 16
5. Conclusion 19
6. Reference 20
7. **Introduction:**

Real-time communication has become a fundamental need in both personal and professional settings, enabling instantaneous exchange of messages, files, and ideas. Traditional HTTP-based methods, while reliable, are limited by their request-response nature, leading to delays and necessitating frequent polling to keep conversations updated. This approach is particularly inefficient in chat applications where users expect real-time interactions. WebSocket technology addresses these limitations by establishing a persistent, two-way connection between the client and server, allowing the server to push updates instantly to connected clients. This capability makes WebSockets ideal for chat applications, where timely delivery of messages is crucial for creating an engaging, seamless user experience. By leveraging WebSockets, our project aims to provide an efficient solution to the demands of real-time communication in chat applications.

The project is a web-hosted chat application built using Flask and WebSocket programming, designed to facilitate both group and private messaging. In the group chat feature, users can send messages to multiple participants simultaneously, making it ideal for team collaborations or community discussions. Furthermore, users in group chats can also share files, which can be any type of document or media. This feature enables collaborative efforts to be more dynamic, as users can exchange documents, images, and other resources instantly. Additionally, the application includes a private chat function where users can initiate direct, one-on-one communication with specific individuals by searching for them through usernames. This private chat option ensures users can exchange both messages and files securely and discreetly, preserving the privacy of their interactions.

A core objective of this project is to establish a robust, secure chat system that supports seamless real-time communication. The choice of WebSocket technology guarantees that messages and files are delivered instantly, maintaining the flow of conversation and enhancing the responsiveness of the chat interface. The application’s design also focuses on flexibility, allowing users to communicate in both group and private settings based on their needs. With real-time updates, users will experience minimal delay in message delivery, which is critical to keeping conversations engaging and interactive.

File sharing is another integral feature of the application, catering to the need for multifaceted communication. Users can exchange files easily, whether in group or private chats, enabling them to share images, documents, and other resources without the need for additional platforms. This functionality enhances the application’s utility in collaborative settings where sharing documents or multimedia files is essential for efficient communication. The seamless integration of file sharing into both group and private chats enriches the user experience by making the application a comprehensive solution for messaging and resource sharing.

1. **Methodology:**

**2.1** **Architecture and Real-Time Communication Model**

This project is a real-time web chat application designed using the Flask framework and WebSocket programming, featuring both group and private messaging with file sharing capabilities. The client-server architecture is implemented with Flask as the core backend framework, managing HTTP requests, user sessions, and file handling, while WebSocket facilitates real-time communication. The server renders HTML pages, prompts for username entry if necessary, and establishes a Socket.IO connection for dynamic messaging. The WebSocket protocol is particularly well-suited for this project, as it allows persistent, bidirectional communication between the client and server. Unlike traditional HTTP, where clients must poll the server for updates, WebSockets enable the server to push data directly to connected clients, making it ideal for the instant, low-latency interactions required by chat applications.

The Flask server initially handles routing and page loads and manages user sessions through a unique username. Upon connecting, each user is prompted to enter a username, which is stored in Flask’s session. Once the username is set, Flask renders the main chat interface, and a WebSocket connection is established to handle real-time events. This connection supports seamless data exchange, where the server can broadcast messages to all connected clients in a group chat or target specific clients for private messages. Flask handles standard HTTP requests, such as serving the initial HTML template, while WebSocket ensures that updates are delivered in real-time, creating a responsive and interactive chat experience.

**2.2** **WebSocket Programming for Real-Time Messaging**

The WebSocket implementation in this chat application enables persistent, low-latency communication, which is essential for real-time messaging. After a WebSocket connection is established via Socket.IO, the server can continuously send messages to clients without the need for repeated HTTP requests. This persistent connection is critical in a chat application, where users expect instantaneous updates and smooth interactions. When a user sends a message, WebSocket broadcasts the message immediately to all participants in a group chat, or directs it specifically to one client in the case of private messaging. WebSocket events, such as `upload\_group\_file` and `upload\_private\_file`, handle file sharing events within chats, allowing the server to distribute shared files directly to the intended recipients. This bidirectional communication allows the chat interface to update immediately whenever a message or file is shared, providing a real-time experience that keeps all participants engaged and connected.

**2.3** **File Uploads and Management**

File sharing is integrated into both group and private chat modes, enhancing the chat application’s functionality by allowing users to exchange documents, images, and other resources. Files are selected through a client-side file input, converted to base64, and transmitted to the server over WebSocket. The base64 encoding ensures that binary data can be safely transported in the WebSocket payload, avoiding potential issues with binary transmission. Upon receiving the encoded file data, the server decodes it and saves it to an `uploads` directory, making the file accessible to users via a generated URL.

To ensure compatibility and security, only specific file types are allowed, such as `.txt`, `.pdf`, `.png`, `.jpg`, `.jpeg`, `.gif`, `.doc`, and `.docx`. The server uses the `allowed\_file()` function to validate that uploaded files match these allowed extensions, thereby preventing unsupported or potentially harmful file types from being processed. Furthermore, filenames are sanitized using Flask's `secure\_filename()` function to prevent path traversal vulnerabilities and ensure file paths are safe and predictable. This function strips dangerous characters from filenames, allowing the server to handle files securely. Once stored, a URL for the file is generated and emitted back to the relevant chat, allowing users to download the file directly.

**2.4** **Security Considerations**

Security is a top priority in this chat application, particularly concerning user session management, file handling, and communication privacy. The application uses Flask’s session management capabilities, powered by a secure secret key, to store user-specific session data, including usernames, ensuring that each user’s chat experience is isolated and protected. This prevents unauthorized access to private chats and files and restricts access to user-specific data, maintaining the privacy and integrity of interactions.

To further secure the application, the `secure\_filename()` function is employed to sanitize filenames before saving them on the server. This mitigates the risk of directory traversal attacks by ensuring that filenames do not contain malicious characters. Additionally, the `allowed\_file()` function limits file uploads to a predefined set of extensions, preventing potentially unsafe files from being uploaded and executed on the server.

Together, these security measures protect against common vulnerabilities and ensure the chat application remains reliable, safe, and user-focused. By combining Flask’s HTTP capabilities with WebSocket’s real-time communication, this application creates a secure, interactive, and responsive platform for both private and group communication.

1. **Implementation:** 
   1. **Frontend Design**

The user interface of the real-time chat application is meticulously designed to support both group and private messaging, incorporating an upload feature to facilitate file sharing. The interface is organized to provide a cohesive user experience, with dedicated sections for group messages, private conversations, and file uploads. Group chat functionality allows all participants within a shared channel to receive real-time messages, fostering a collaborative communication environment. For private conversations, users may initiate direct messaging by selecting a specific recipient using their unique username.

A distinct upload section enhances usability by allowing users to attach and send files. This feature integrates a progress bar to visually track upload progress, improving user awareness and transparency in file transfers. Furthermore, visual cues are provided for both message and file types, distinguishing between text and file messages to ensure clarity and usability. The user interface, styled with CSS, follows responsive design principles to optimize accessibility across devices and platforms, reinforcing an interactive and consistent communication experience.

* 1. **Backend Implementation**

The backend of the chat application is structured around Flask for HTTP handling, session management, and WebSocket-based real-time communication. Flask initializes user sessions and manages routing for all HTTP requests associated with rendering the chat interface and handling user inputs. Upon user entry, the server prompts the individual to input a unique username, stored within Flask’s session to enable user identification across both group and private chats. Flask’s routing and session management thus serve as essential components in establishing a unique identity for each user.

To enable real-time interaction, WebSocket (implemented via Flask-SocketIO) creates a persistent and bidirectional communication channel between the client and server. This structure is critical for maintaining the application’s responsiveness, as WebSocket events facilitate instant broadcasting of messages and file data to multiple users. For each session, the server uses WebSocket to listen for various events (such as message and file transmissions) and respond accordingly, broadcasting information to relevant users based on the event type. The combined use of Flask for session handling and WebSocket for real-time communication thus establishes a dynamic environment suited to interactive messaging.

* 1. **Code Walkthrough**

The code is structured to integrate Flask’s session management and routing capabilities with WebSocket’s event-based communication model. This combination supports the application’s core functions, such as handling real-time messages, managing user sessions, and facilitating secure file sharing. Key aspects of the code include session management for user tracking, WebSocket event handling for messaging and file transmission, and file upload handling with validation.

* + 1. **Session Management**

Flask’s session management framework is instrumental in identifying and maintaining user states across sessions. Each user must provide a unique username upon accessing the chat application, which is stored in the Flask session object. This session-based tracking differentiates users within group and private chat contexts, enabling the server to direct messages and file data to intended recipients. Secure session identifiers are assigned to ensure data integrity and privacy, thereby safeguarding user information and interactions within the application.

* + 1. **Event Handling for Messages and Files**

The WebSocket protocol is essential for supporting the real-time requirements of the application through event-based communication. Primary WebSocket events handle the real-time transfer of both messages and file data, with distinctions made between group and private communication channels:

* **Group Messaging Events**: The message event in the code is configured to broadcast messages to all users within a group chat. When a client initiates a message event, the server, using socket.emit, broadcasts the message to every active user in the group. This enables seamless, real-time group interaction, ensuring that all participants remain synchronized.
* **Private Messaging Events**: For private messages, WebSocket enables direct message delivery by associating each recipient with a unique session ID. This targeted approach ensures that only the designated recipient receives private messages, maintaining confidentiality and privacy within one-on-one communications.
* **File Upload Events**: Separate WebSocket events handle file uploads, managing real-time file transmission for both group and private contexts. Upon selecting a file, the client reads it in base64 format and sends it via WebSocket. The server then verifies and broadcasts the file URL to either the group or a specific private recipient, depending on the event type. This setup allows for efficient real-time file sharing within an interactive environment.
  1. **File Upload Handling and Validation**

File handling and validation are integral to maintaining security and functionality within the chat application’s file-sharing feature. Users can initiate file uploads through the frontend interface, where selected files are encoded in base64 and transmitted via WebSocket. A progress bar on the client side provides users with real-time feedback during upload. Upon receiving the file data, the server validates the file’s extension using an allowed\_file() function to confirm compatibility with permitted file types. Flask’s secure\_filename() function is applied to sanitize filenames, minimizing risks associated with unauthorized file access or directory traversal attacks.

Files are stored on the server in a dedicated uploads directory, with unique URLs generated for each uploaded file. These URLs are then emitted to the appropriate chat, enabling users to access and download files directly. By implementing a rigorous validation process and secure file handling mechanisms, the application maintains a robust file-sharing capability that prioritizes user safety and data integrity.

* + 1. **Key Code Sections**
* **@socketio.on('upload\_group\_file')**: This section of the code handles group file uploads. When a user initiates an upload in a group chat, the server receives the file data, validates it, and stores it in a secure directory. A unique URL is generated for each file, which is then broadcasted to all users in the group via WebSocket, enabling immediate access.
* **@socketio.on('upload\_private\_file')**: This decorator function manages private file uploads. The server uses the recipient’s username and session ID to securely route the uploaded file to the intended recipient only. The file data is validated and stored similarly to group files, but the URL is emitted solely to the designated user. This approach ensures that private file transfers remain secure and user-specific.

**3.5 Methods:**

**1. register\_user(username):**

This function handles the registration of a user when they first connect to the SocketIO server. The user's session ID (request.sid) is stored in a dictionary (users) with their username as the key.

Purpose:

When a user connects, we want to associate them with a session so that we can identify them and send messages (either group or private) specifically to them.

How it works:

* The username is passed as a parameter via the register event.
* The session ID (request.sid) is stored in a global users dictionary, where the username is the key and the session ID is the value.
* The print statement outputs the username and their corresponding session ID to the console.

**2. handle\_disconnect():**

This function is triggered when a user disconnects from the server. It removes the user's session from the users dictionary.

Purpose:

To clean up and ensure that users are properly removed from the active list when they disconnect.

How it works:

* It loops through the users dictionary, checking for the user whose session ID matches request.sid (the current session ID).
* Once a matching user is found, that user's entry is removed from the users dictionary.
* A print statement outputs that the user has disconnected.

**3. handle\_group\_message(data):**

This function handles sending group messages to all connected users.

Purpose:

To broadcast a message to all connected users in a "group" chat setting.

How it works:

* The sender's username is retrieved from the session (session["username"]).
* The message data is passed via the data argument, and the message itself is accessed with data['message'].
* The emit function is used to send the message to all connected clients by broadcasting it using broadcast=True.

**4. handle\_private\_message(data):**

This function handles sending private messages between users.

Purpose:

To send a direct message to a specific user (private chat).

How it works:

* The sender's username is retrieved from the session (session["username"]).
* The recipient is specified in the data['recipient'] field.
* If the recipient is found in the users dictionary, the message is sent to them using emit to their specific session (identified by their sid).

**5. handle\_group\_file\_upload(data):**

This function handles the uploading of files to a group chat.

Purpose:

To upload files that can be shared with all users in the group chat.

How it works:

* The file's name and data are passed in the data argument.
* The file data is base64 decoded using base64.b64decode(data['fileData']).
* The decoded file data is saved to the server's file system in the specified UPLOAD\_FOLDER.
* The file URL is generated using Flask's url\_for, which constructs a URL for accessing the uploaded file.
* A message containing the file's name, URL, and the sender's username is broadcast to all connected clients using emit.

**6. handle\_private\_file\_upload(data):**

This function handles the uploading of files for private messaging between two users.

Purpose:

To send files privately to a specific recipient.

How it works:

* Similar to handle\_group\_file\_upload, the file's name and data are passed in the data argument.
* The file is base64-decoded and saved to the server.
* A URL is generated for accessing the uploaded file.
* The message containing the file’s name, URL, and the sender’s username is sent to the recipient (identified by their session ID).
* The file message is also sent back to the sender (for confirmation).

**7. uploaded\_file(filename):**

This is a Flask route that serves the uploaded file.

Purpose:

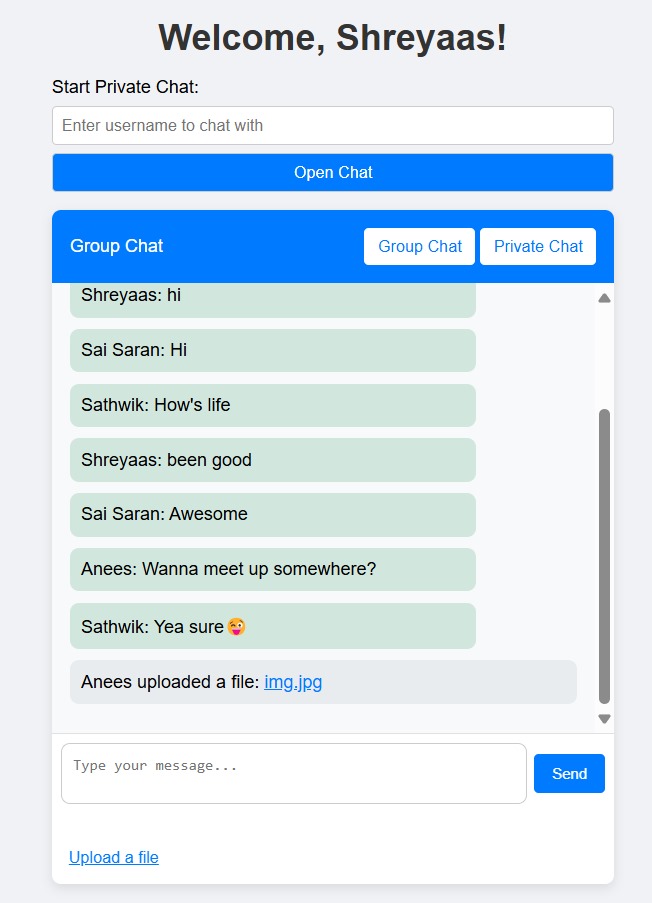
To serve the uploaded file to clients who request it.

How it works:

* This route uses Flask's send\_from\_directory to send the file from the UPLOAD\_FOLDER to the client when the URL is requested.
* The filename is extracted from the URL and used to find the file in the UPLOAD\_FOLDER.

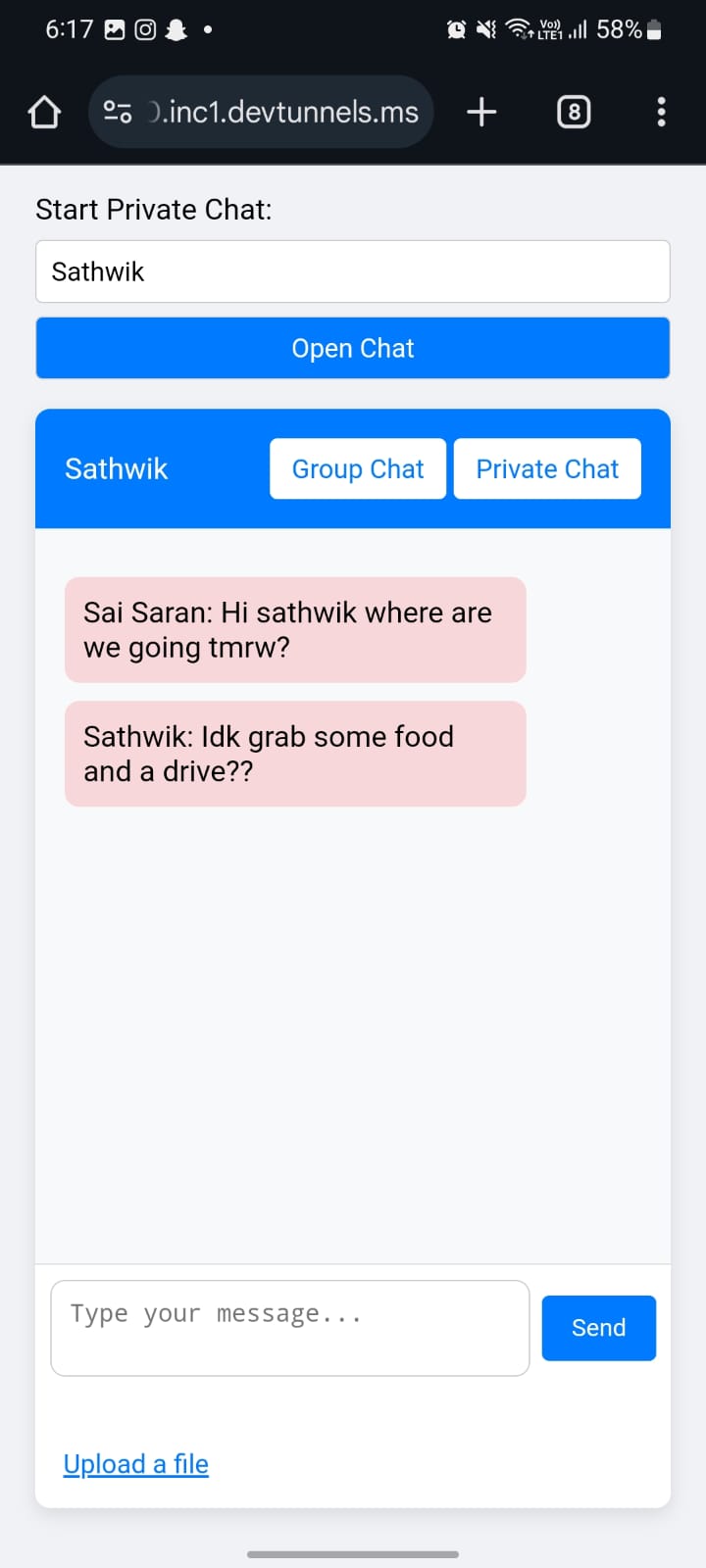
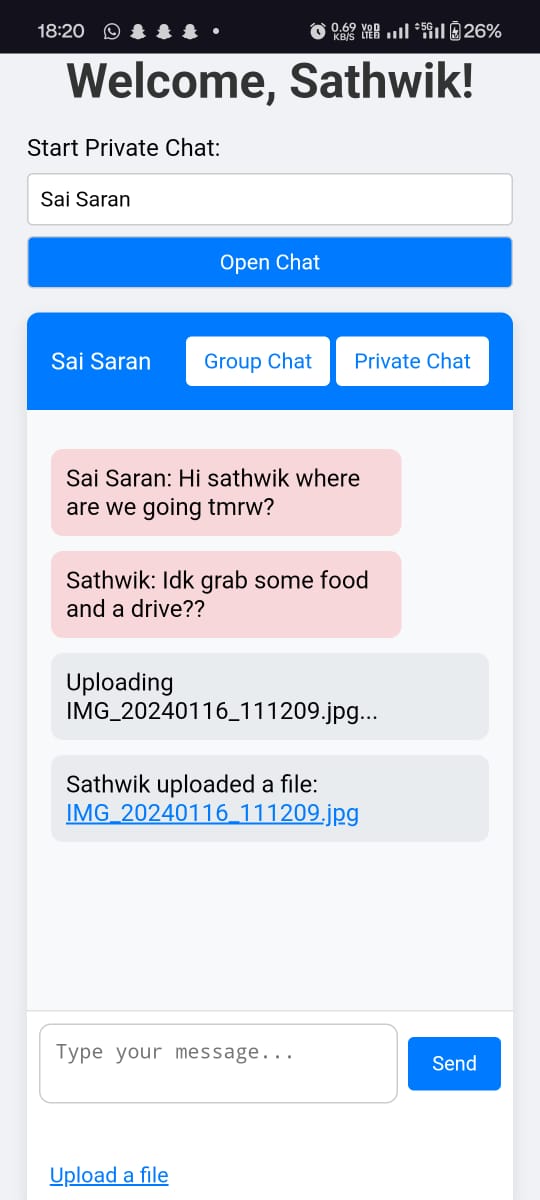
1. **Results and Discussions:**

The implementation of the real-time chat application was tested in various scenarios to verify its functionality, responsiveness, and reliability. Screenshots from the testing phase are provided below to illustrate key features, including group messaging, private messaging, file sharing, and upload progress tracking. Each screenshot demonstrates the application’s performance and user experience, highlighting both design and functionality aspects.



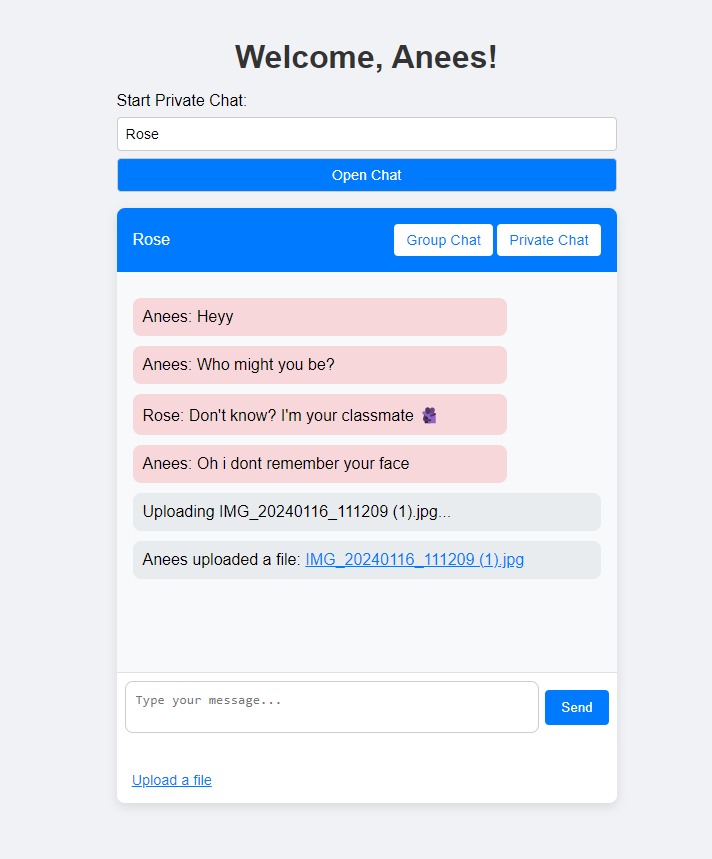
**Figure 1**

The Group Chat Interface (Figure 1) showcases the layout of the chat area, where multiple users can communicate in real-time. Messages sent by any user are instantly broadcasted to all participants, reflecting the WebSocket-based real-time updates. The consistent delivery and reception of messages among all users confirm that the group chat functionality is working as expected.



**Figure 2**

The Private Messaging Interface (Figure 2) illustrates a one-on-one conversation. In this scenario, messages are directed solely to the specified recipient, as verified by the presence of user-specific session identifiers. This functionality ensures that private messages remain visible only to the designated recipient, supporting secure, individualized communication.



**Figure 3**

Additionally, File Upload and Sharing (Figure 3) was tested with a variety of file types permitted by the application’s file validation function. As seen in the screenshot, users can upload files directly from their device, triggering the progress bar to indicate upload status. Once uploaded, the server generates a secure URL, which is instantly shared in the chat, allowing recipients to download or view the file. This functionality was tested in both group and private chat modes to confirm that files are accurately shared in real-time to the intended users.

These results affirm the successful integration of Flask and WebSocket functionalities to deliver a reliable, responsive chat experience. The screenshots provide a visual verification of the implemented features and demonstrate the application’s ability to manage real-time, secure communication and file sharing efficiently

1. **Conclusion:**

In conclusion, this chat application successfully demonstrates the power of Flask and WebSocket programming for creating a real-time communication platform. By implementing both group and private chat functionalities, along with the ability to share files, the application provides a comprehensive solution for online interactions. The use of WebSockets allowed for instant communication between users, ensuring messages and files were transmitted with minimal delay. The file-sharing feature, which supports various file types, further enhances the usability of the application, making it a valuable tool for both personal and professional use.

Throughout development, challenges such as managing file uploads efficiently and ensuring seamless communication across multiple users were encountered. These were addressed using solutions like base64 encoding for file transmission and sanitizing filenames to prevent security vulnerabilities. The application was rigorously tested, and it performed as expected, with multiple users able to participate in group chats, send private messages, and share files without issues.

However, while the application functions well for small-scale usage, there is room for improvement when scaling to a larger audience. Enhancements in server resource management, particularly for handling larger file uploads and concurrent users, would be necessary for broader adoption. Additionally, adding security measures like message encryption and user authentication would further protect user data and ensure privacy. Expanding the user interface to include features like chat history, notifications, or real-time presence indicators could also improve the overall experience.

Looking ahead, this project lays the groundwork for more advanced real-time web applications. With the knowledge gained from using Flask and WebSockets, future versions could incorporate features like voice or video messaging, as well as enhanced scalability to support a growing user base. Overall, this project highlights the potential of combining Flask with real-time communication technologies to build dynamic, interactive, and secure web applications.

1. **Reference:**
   1. <https://codeburst.io/building-your-first-chat-application-using-flask-in-7-minutes-f98de4adfa5d>
   2. <https://thepythoncode.com/article/how-to-build-a-chat-app-in-python-using-flask-and-flasksocketio>
   3. <https://github.com/msindev/Chat-App-Flask-SocketIO/blob/main/app.py>