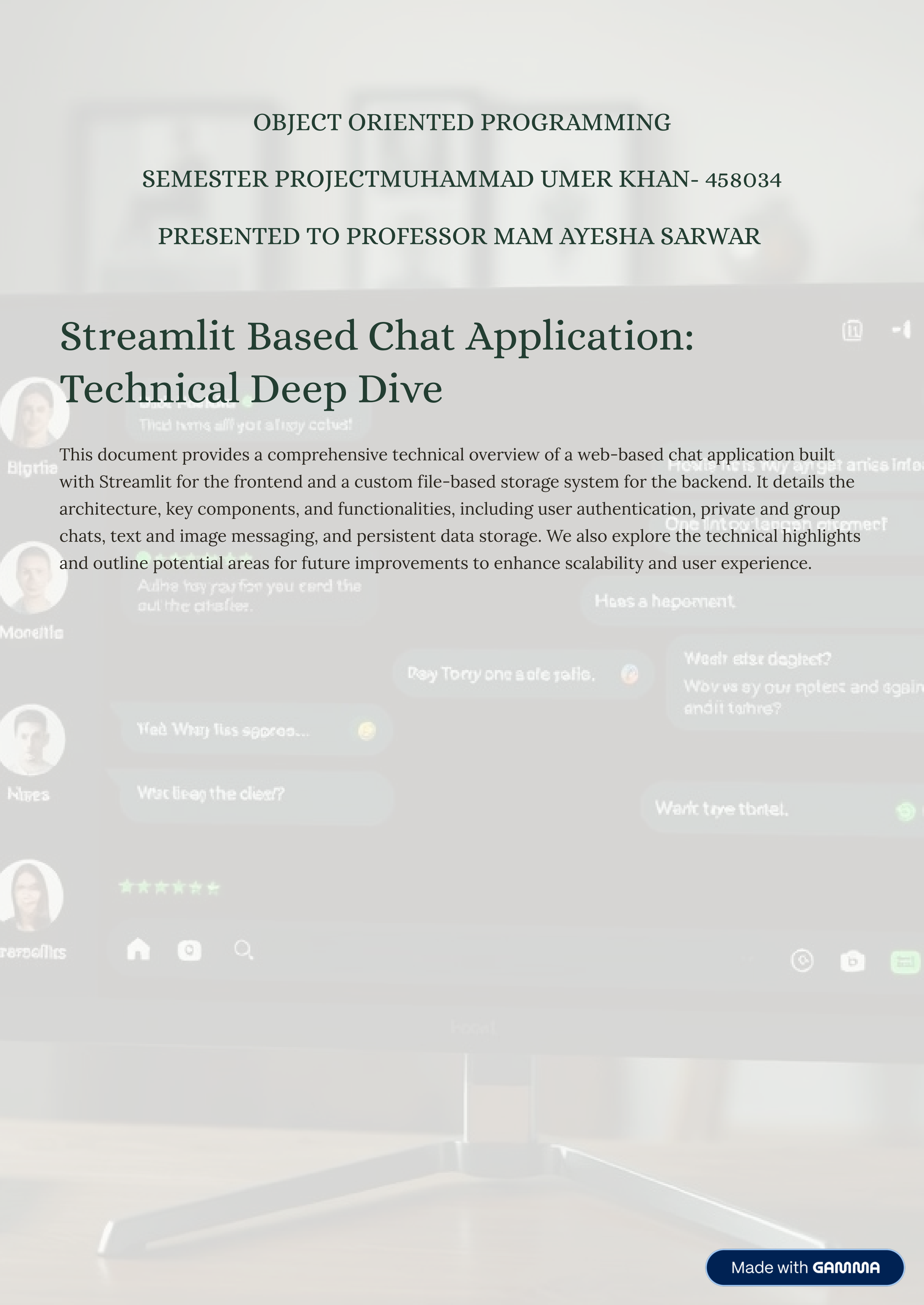
OBJECT ORIENTED PROGRAMMING

SEMESTER PROJECTMUHAMMAD UMER KHAN- 458034

PRESENTED TO PROFESSOR MAM AYESHA SARWAR

Streamlit Based Chat Application: Technical Deep Dive



This document provides a comprehensive technical overview of a web-based chat application built with Streamlit for the frontend and a custom file-based storage system for the backend. It details the architecture, key components, and functionalities, including user authentication, private and group chats, text and image messaging, and persistent data storage. We also explore the technical highlights and outline potential areas for future improvements to enhance scalability and user experience.

Application Overview & Core Features

The Streamlit Chat Application is a robust, web-based platform designed for real-time communication. It leverages Streamlit's intuitive framework for a responsive frontend user interface and implements a custom backend storage layer for data persistence. This architecture ensures a seamless and interactive user experience while maintaining data integrity.

User Authentication

Secure sign-up and login mechanisms to manage user access, including username availability checks and password verification.

Private & Group Chats

Supports one-on-one direct messages (DMs) and multi-user group conversations, providing versatile communication options.

Text & Image Messaging

Enables users to send both text-based messages and image files, enriching communication capabilities.

Automatic Message Refresh

Messages are automatically refreshed every 3 seconds, ensuring near real-time updates for an engaging chat experience.

Persistent Data Storage

All user, chat, and message data are persistently stored using JSON files, making the application portable and easy to debug.

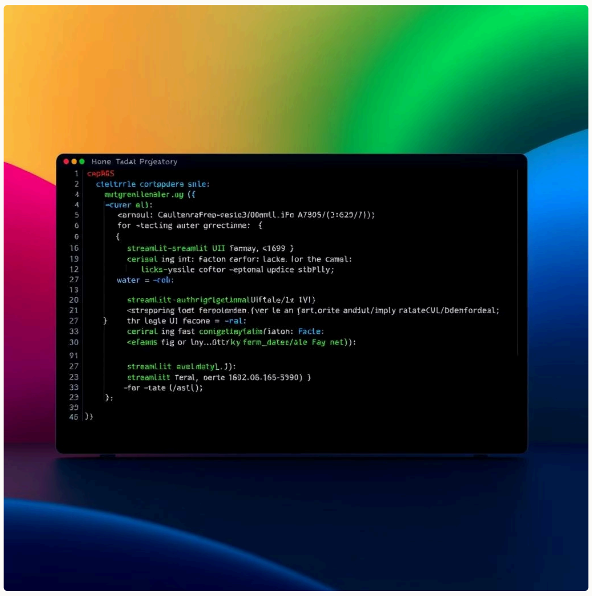
The application's design prioritizes a straightforward setup and operation, making it ideal for developers and product managers seeking a clear and functional example of a Streamlit-based chat system with a custom backend.

Frontend Architecture: app.py

The app.py script serves as the primary interface for the Streamlit Chat Application, managing all user interactions, session states, and the dynamic display of chat content. It is engineered to provide a fluid and intuitive user experience, from initial authentication to active messaging.

Key Components & Features

- **Streamlit UI Configuration:** Utilizes `st.set_page_config(page_title="Umer Chat", layout="wide")` to establish a wide layout and a custom title, optimizing the visual presentation across various screen sizes.
- **User Authentication (`login_view()`):** Features a tabbed interface for both login and sign-up. The login tab verifies existing credentials, while the sign-up tab facilitates new account creation, including robust password confirmation and real-time username availability checks.
- **Sidebar Navigation:** A persistent sidebar displays the currently logged-in username, provides a clear option to log out, and includes a toggle for the auto-refresh feature. It dynamically lists all chats associated with the user and offers functionalities to initiate new direct messages or group chats.
- **Chat View:** The main content area dynamically displays the selected chat's name, its participants, and the chronological stream of messages. Each message is rendered with a timestamp, sender's name, and optionally, an embedded image.
- **Message Composition & Auto-Refresh:** An integrated text input box and image uploader allow users to compose and send messages seamlessly. The auto-refresh mechanism, powered by a `time.sleep(3)` and `st.rerun()` loop, ensures chat content is updated approximately every three seconds, providing a near real-time communication experience.
- **Message Rendering (`render_message()`):** This dedicated function is responsible for the precise formatting and display of individual messages, accommodating both text and image content for a rich chat display.



Main Execution Flow

The application's core logic hinges on a conditional flow: it first verifies the user's login status. If authenticated, it proceeds to load the sidebar and the currently selected chat, providing access to the full chat functionality. Conversely, if the user is not logged in, the application intelligently presents the login and sign-up forms, guiding new or returning users through the authentication process before granting access to the chat features.

This modular design in app.py ensures a clean separation of concerns, making the frontend logic maintainable and scalable, while delivering a highly interactive and user-friendly chat environment.

Backend Storage Layer: storage.py

The `storage.py` module is the backbone of the Streamlit Chat Application, serving as the persistent data layer. It meticulously manages user accounts, chat conversations, and message content, employing a file-based storage system using JSON files. This approach ensures data durability and facilitates straightforward debugging and portability.

Directory Structure

- `data/users/`: Dedicated to storing individual user account files. Each user's data, including hashed passwords and associated metadata, resides here.
- `data/chats/`: Contains all chat conversation data. Each chat, whether private or group, is stored as a separate JSON file, encapsulating messages and participant information.
- `data/images/`: Serves as the repository for all uploaded image files, ensuring that multimedia content sent within chats is persistently stored and retrievable.

Security Measures

> A critical aspect of `storage.py` is its robust security implementation for user passwords. Passwords are never stored in plaintext. Instead, they undergo a cryptographic transformation: they are combined with a unique salt for each user and then hashed using the SHA-256 algorithm. This salting and hashing process effectively mitigates risks such as rainbow table attacks and brute-force attempts, significantly enhancing user data security.

Core Backend Functionalities

- **User Management:**
  - `list_users()`: Retrieves a comprehensive list of all registered usernames.
  - `user_exists()`: Verifies the existence of a specific username.
  - `create_user()`: Registers new users, securely storing their credentials after hashing and salting their passwords.
  - `verify_user()`: Authenticates user login attempts by validating provided credentials against stored hashed passwords.
  - `get_user()`: Fetches detailed information for a specified user.
- **Chat Management:**
  - `create_private_chat()`: Establishes direct message chats, assigning unique IDs based on the participating usernames to ensure distinct conversations.
  - `create_group_chat()`: Initializes group chats, generating a random unique ID for each group.
  - `list_user_chats()`: Compiles and returns a list of all chat conversations a given user is part of.
  - `get_chat()`: Retrieves the complete details of a specific chat, including all messages and participants.
  - `append_message()`: Integrates new messages into the respective chat's data file.
- **Image Handling:**
  - `save_image_bytes()`: Stores uploaded images directly to the designated `data/images/` directory.
  - `image_to_data_url()`: Converts image files into Base64 encoded data URLs, enab
    embedding within HTML for display in the frontend

Key Technical Highlights

The Streamlit Chat Application incorporates several strategic technical decisions that underpin its functionality and user experience. These highlights demonstrate a pragmatic approach to building a real-time chat system with a focus on simplicity, security, and responsiveness.

1

Stateless Session Management

The application leverages Streamlit's built-in `st.session_state` to maintain critical user information, such as the logged-in user's identity and the currently active chat. This approach ensures that user-specific data persists across reruns without relying on complex backend session handling, simplifying the state management logic significantly.

2

Pseudo Real-time Auto-Refresh

Near real-time updates are achieved through a polling mechanism that combines `time.sleep(3)` with `st.rerun()`. Every three seconds, the Streamlit application reruns, fetching the latest messages and updating the chat view. While not a true WebSocket implementation, this method provides a sufficiently responsive experience for many use cases, maintaining simplicity in the architecture.

3

Human-Readable File-Based Persistence

All application data—including user profiles, chat histories, and message content—is stored in human-readable JSON files. This file-based persistence offers significant advantages for debugging, as developers can easily inspect and modify data directly. It also makes the application highly portable, requiring no external database setup and runnable on any system with Python and Streamlit installed.

4

Robust Password Security

Security is a cornerstone of user authentication. The application implements a strong password security strategy by combining unique salts with SHA-256 hashing for all user passwords. This prevents the storage of plaintext passwords and significantly enhances protection against common cryptographic attacks, such as rainbow table lookups.

5

Dynamic UI with Streamlit Widgets

Streamlit's rich ecosystem of widgets—including `st.selectbox`, `st.form`, and `st.file_uploader`—is extensively utilized to create a highly interactive and dynamic user

6

Cross-Platform Compatibility

A key advantage of this application's design is its inherent cross-platform compatibility. By foregoing external datab

dependencies and relying solely on Python

In-depth: User Authentication Flow

User authentication is a critical component of the Streamlit Chat Application, ensuring that only authorized users can access their private and group conversations. The system provides both sign-up and login functionalities, designed for security and ease of use.

1

Sign-Up Process

- Users access the "Sign-up" tab within the `login_view()`.
- They provide a desired username and set a password, which must be confirmed.
- The system performs an immediate check using `user_exists()` to ensure the username is unique.
- If available, `create_user()` is invoked to register the new account.
- The password is salted and SHA-256 hashed before being stored in `data/users/`.
- A unique salt is generated for each user, preventing identical passwords from having identical hashes.

2

Login Process

- Users navigate to the "Login" tab in `login_view()`.
- They input their username and password.
- The `verify_user()` function retrieves the stored hashed password and salt for the given username.
- The entered password is then hashed with the retrieved salt.
- The newly generated hash is compared against the stored hash.
- Upon successful verification, the user's details are loaded into `st.session_state`, granting access to the application.

This dual-tabbed authentication interface simplifies the entry point for both new and returning users, while the robust backend security measures ensure that user credentials remain protected.

Chat Management & Messaging Capabilities

The Streamlit Chat Application provides comprehensive functionalities for managing both private and group conversations, allowing users to send text and image messages. The core logic for these features resides within the `storage.py` module, which handles the creation, retrieval, and updating of chat data.

Creating Chats

- **Private Chats (`create_private_chat()`):**
  - Initiated between two users.
  - A unique chat ID is generated by concatenating and hashing the usernames of the two participants (e.g., `user1_user2_hashed_id`), ensuring that a specific DM chat between two individuals always has the same identifier, regardless of who initiated it.
  - This approach prevents the creation of duplicate private chats between the same pair of users.
- **Group Chats (`create_group_chat()`):**
  - Designed for multi-user conversations.
  - A randomly generated unique ID is assigned to each new group chat, allowing for distinct group instances.
  - Users can add multiple participants during creation, and the chat details, including participant lists, are stored persistently.

Message Sending & Display

- **Text Messages:** Users compose messages in a dedicated text area, which are then appended to the relevant chat's JSON file using `append_message()`. Each message includes the sender's name, content, and a timestamp.
- **Image Messages:**
  - The `st.file_uploader` widget allows users to select and upload image files (e.g., PNG, JPEG).
  - The uploaded image bytes are saved to the `data/images/` directory via `save_image_bytes()`.
  - For display in the frontend, the image file path is converted into a Base64 data URL using `image_to_data_url()`. This embedding method allows images to be rendered directly within the Streamlit UI without requiring a separate server for image serving.
- **Automatic Refresh:** Messages in the active chat view are automatically updated every 3 seconds through a `st.rerun()` call, providing a dynamic and responsive chat experience. This ensures new messages from other participants appear promptly.

This comprehensive message and chat management system provides a solid foundation for interactive communication within the application.

Data Persistence and Security Deep Dive

The Streamlit Chat Application's reliance on file-based JSON storage for all its data offers unique advantages in terms of portability and debuggability, while its security measures ensure user data protection.

File-Based Persistence (JSON)

All application data is stored in plain text JSON files within a structured `data/` directory. This design choice provides a transparent and accessible data layer:

- **User Data (`data/users/`):** Each user has a dedicated JSON file containing their username, the salted and hashed password, and potentially other metadata. This granular storage facilitates easy management and retrieval of individual user profiles.
- **Chat Data (`data/chats/`):** Every private and group chat is represented by its own JSON file. These files store an array of messages, with each message object containing details like sender, timestamp, message content (text), and image path (if applicable). This structure makes it straightforward to reconstruct chat histories.
- **Image Data (`data/images/`):** Uploaded images are stored directly as binary files. Their paths are referenced within the chat JSON files, and they are converted to Base64 on demand for display in the frontend.

The simplicity of JSON files means developers can inspect or even manually edit data directly, making development and debugging highly efficient. Furthermore, the absence of an external database makes deployment trivial, as the entire application can be moved by simply copying the project directory.

Robust Password Security

User password security is paramount, and the application employs a multi-layered approach to protect sensitive credentials:

- **Salting:** When a new user signs up, a unique, randomly generated salt is created for their password. This salt is combined with the user's plaintext password before hashing. The primary benefit of salting is to defend against rainbow table attacks, as even if two users choose the same password, their unique salts will result in different hashed values.
- **Hashing (SHA-256):** After salting, the combined string (password + salt) is subjected to the SHA-256 cryptographic hash function. SHA-256 is a one-way function, meaning it's computationally infeasible to reverse the hash to obtain the original password. This ensures that even if the hashed password database is compromised, the actual passwords remain secure.
- **No Plaintext Storage:** Crucially, no plaintext passwords are ever stored, neither in memory for longer than necessary during the hashing process nor on disk. Only the hashed and salted versions are retained, significantly minimizing the risk of direct password exposure.

This combination of salting and strong hashing algorithms provides a robust defense against common password cracking techniques, ensuring a secure authentication system for the chat application.

Future Enhancements & Scalability

While the current Streamlit Chat Application provides a robust and functional foundation, several potential improvements could significantly enhance its scalability, performance, and user experience. These enhancements aim to address the limitations of a file-based system and introduce advanced real-time functionalities.

Migrate to a Database for Storage

The current JSON file-based storage, while simple and portable, can face scalability challenges with a large number of users or extensive chat histories. Migrating to a structured database system would offer substantial benefits:

- **Relational Databases:** Options like **SQLite** (for lightweight, embedded needs) or **PostgreSQL** (for more robust, multi-user deployments) would provide ACID compliance, better indexing for faster data retrieval, and native support for complex queries.
- **NoSQL Databases:** For higher flexibility and horizontal scalability, NoSQL databases like MongoDB could be considered, especially if the data model becomes more dynamic.
- **Benefits:** Improved read/write performance

Implement WebSockets for Real-Time Updates

The current auto-refresh mechanism relies on polling (`time.sleep(3)` and `st.rerun()`), which is inefficient and generates unnecessary server load. A WebSocket-based solution would provide true real-time communication:

- **Mechanism:** WebSockets establish a persistent, full-duplex communication channel between the client and server. Instead of polling, the server can push new messages directly to clients as soon as they are sent.
- **Frameworks:** Integrating a Python WebSocket library (e.g., `websockets`, `SocketIO`) with Streamlit (perhaps running a separate backend service) would enable instant message delivery.
- **Benefits:** Lower latency for message delivery, reduced server load (no

Enhance User Experience Features

Several features common in modern chat applications could significantly improve the user experience:

- **Message Search Functionality:** Allow users to search for specific keywords within their chat histories, requiring efficient indexing of message content, ideally supported by a database.
- **User Typing Indicators:** Provide visual cues (e.g., "User X is typing...") to indicate when another user is composing a message. This would necessitate a real-time communication channel like WebSockets.
- **Read Receipts:** Implement a mechanism to show when messages have been read by recipients, adding a layer of ackno conversations. This

Conclusion & Recommendations

The Streamlit Chat Application, as designed, offers a solid foundation for real-time communication with its intuitive Streamlit frontend and robust file-based backend. It successfully demonstrates key functionalities like user authentication, diverse chat types, and persistent messaging, all while emphasizing security through salted and hashed passwords.

For immediate deployment and evaluation, the current architecture is highly effective due to its simplicity, portability, and minimal dependencies. It serves as an excellent reference for developers and product managers seeking to understand or extend Streamlit's capabilities in building interactive web applications.

Prioritize Database Migration

For any production-level deployment or expected growth in user base and data volume, migrating from JSON files to a relational database (e.g., PostgreSQL) should be the **first priority**. This will drastically improve scalability, data integrity, and query performance.

Integrate WebSockets

To achieve true real-time responsiveness and enable features like typing indicators and read receipts, invest in a WebSocket-based communication layer. This will eliminate inefficient polling and provide a much smoother user experience.

Enhance UX with Advanced Features

Once the core infrastructure is optimized, focus on implementing user-centric features like message search, read receipts, and typing indicators to bring the application closer to modern chat standards.

Regular Security Audits

While strong password hashing is in place, continuous security audits and updates to cryptographic practices are recommended to protect against evolving threats.

By focusing on these strategic improvements, the Streamlit Chat Application can evolve into a highly scalable, feature-rich, and robust communication platform capable of supporting a larger user base with an enhanced real-time experience.