System Design and Setup Documentation for Chatly

A real-time messaging service prototype.

1. Project Name: Chatly

Objective: Build a messaging service prototype that allows users to send and receive text messages and files, create group chats, and manage real-time message updates.

1.1 Project Overview

Chatly is a messaging service prototype designed to enable users to send and receive text messages, share files, create group chats, and personal chats with other users, and receive real-time message updates. The core features include user registration and authentication, text messaging, group chat functionality, and real-time message updates. The system is developed using Django as the backend with WebSockets via Django Channels for real-time functionality, and HTML, CSS, JS, and React.js for the front end.

1.2 Core Features:

- User registration and authentication
- Real-time messaging
- Private and group chat functionality
- File sharing in chats
- User profile management
- Online user status tracking

2. Architecture Overview

The system architecture follows a client-server model using Django as the backend framework and HTML, CSS, and JavaScript for the front end. WebSockets and Django Channels provide real-time communication. The SQLite database stores user and message data. Daphne serves as the ASGI server, and Redis is used as the channel layer for managing WebSocket connections.

2.1 High-Level Architecture

This system is composed of several core components:

2.1.1. Frontend: CSS, HTML, and JavaScript are used to create an interactive user interface.
2.1.2. Backend: Django and Django REST Framework for managing user authentication, group chats, and storing messages, Socket.io.
2.1.3. Real-time Communication: Django Channels and WebSockets facilitate real-time text messaging and file sharing.
2.1.4. Database: SQLite (default database provided by Django) is used to store user data, chat messages, and group information.
2.1.5. Redis: Used as the channel layer backend for handling WebSocket connections.

2.2 Detailed Architecture

2.2.1. User Registration and Authentication: Handled by Django, with support for secure authentication, including optional "Remember Me" functionality.

2.2.2. Sending/Receiving Messages: Real-time text and file messages between users are managed using WebSockets with Django Channels.

2.2.3. Group Chat Functionality: Users can create a new group, assign a name, and invite members. Group admins can edit group info and remove/add members, while other members can leave the group.

2.2.4. Real-Time Messaging Updates: WebSockets and HTMX are used to push new messages to users instantly.

2.2.5. Online Status Indicator: Users can see the online/offline status of other users, marked with a green or gray dot as well as the count of the number of online users.

2.2.6. File Sending: Users can send files of all types within the chat (e.g., images, PDFs).

2.2.7. Chat Interface: Once logged in, users land on a default public chat room. A drop down allows users to switch between group and private chats.

3. Database Schema

The following are the key tables used in the database to manage user authentication, messages, group chat, and permissions:

3.1. a_rtchat_chatgroup: Stores information about chat groups, including the group name, admin, and privacy settings.

3.2. a_rtchat_chatgroup_members: Stores the group membership information.

3.3. a_rtchat_chatgroup_users_online: Tracks the online status of group members.

3.4. a_rtchat_groupmessage: Stores messages sent in group chats, including file attachments.

3.5. a_users_profile: Stores user profile information, including display name and avatar.

3.6. Other authentication and permissions tables (auth_user, auth_group, etc.).

4. System Workflow

4.1. On the first visit, users see a login/signup screen with a welcome message.

4.2. After logging in, the user is directed to a public chat by default.

4.3. Users can initiate private chats from the user profile or join group chats.

4.4. Group chat creators can manage members, edit group details, and remove users. Group members have the option to leave the group.

4.5. All messages are sent and received in real-time using HTMX and WebSockets.

4.6. Users can also send files, and their online status is shown via a dot next to their avatar (green for online, gray for offline).

4.7. Users can change their profile information (display name, avatar) from the profile section.

5. Libraries and Dependencies

The following libraries and dependencies are used in the Chatly project:

5.1. Django: Backend web framework, used for user management and REST APIs.

5.2 Django Channels: Manages WebSocket connections for real-time messaging.

5.3 Redis: Required by Django Channels for message passing.

5.4 HTMX: Handles dynamic loading of messages and real-time updates without page reloads.

5.5 SQLite: The database used to store user and chat information.

5.6 Socket.io: Real-time communication library.

5.7 Websockets: For bi-directional communication between the client and server.

Other dependencies are listed in the requirements.txt file.

6. Setup Instructions

6.1. Clone the repository:

```
git clone https://github.com/Mayur-143/Chatly.git

cd Chatly
```

6.2. Setup a virtual environment and activate it:

```
python3 -m venv venv

source venv/bin/activate    # For Windows use: venv\Scripts\activate
```

6.3. Install the required dependencies:

```
pip install -r requirements.txt
```

6.4. Run database migrations:

python manage.py makemigrations

python manage.py migrate

6.5. Run the Django development server:

python manage.py runserver
Access the application by visiting http://127.0.0.1:8000 in your browser.

6.6. Test the WebSocket functionality by sending messages between users.

7. Design Choice

Here are the reasons why each of the chosen technologies fits well with the Chatly messaging service prototype:

7.1. Django: Django is a powerful web framework that follows the Model-View-Template (MVT) architecture. It is well-suited for building web applications with integrated user authentication, real-time messaging, and group chat functionalities.

Advantages:

- Built-in features for user authentication (via Django-allauth).
- Rapid development with less boilerplate code.
- Excellent ORM for working with databases like SQLite, which simplifies database management.

7.2. Django Channels: Django Channels is an extension of Django that adds support for handling WebSockets, enabling real-time features like messaging updates and user presence tracking.

Advantages:

- Real-time capabilities for messaging (group chat, private chat).
- Scalable WebSocket handling to support multiple users in real time.
- Perfect fit for asynchronous communication, which is needed for live messaging and real-time notifications.

7.3. WebSockets: WebSockets allow for full-duplex communication between the server and the client, enabling real-time messaging.

Advantages:

- Efficient communication with low overhead for real-time data transmission.
- Immediate updates of messages, online statuses, and file-sharing activities.
- Reduces the need for constant polling, thus lowering server load and response time.

7.4. SQLite (Database): SQLite is a lightweight, serverless, and easy-to-configure database, making it ideal for prototyping or smaller applications.

Advantages:

- Zero-configuration and fast setup.
- Suitable for rapid development and testing environments.
- Ensures efficient data storage for managing users, chats, messages, and user profiles.

7.5. HTMX: HTMX allows dynamic, real-time interaction in the front end without needing to rely on complex JavaScript frameworks.

Advantages:

- Simplifies front-end interactions by making AJAX requests directly from HTML.
- Can handle message updates in real time without full-page reloads.
- A lightweight alternative for real-time updates compared to JavaScript-heavy frameworks.

7.6. Django REST Framework: DRF is used to build RESTful APIs, which allows for easy interaction between the front-end and the back-end of the application.

Advantages:

- Simplifies building APIs for user authentication and data exchange.
- Offers serializers that map between Django models and JSON format, ensuring efficient data handling.
- It integrates seamlessly with Django and is widely used in production-ready applications.

7.7. Redis (Optional for WebSockets Management): Redis is used with Django Channels for managing WebSocket connections and efficiently handling channel layers and message brokering.

Advantages:

- Acts as an in-memory store to manage real-time message queues.
- Ensures fast, low-latency message delivery.
- Allows for scaling when multiple users participate in group chats.

7.8. Daphne: Daphne is the ASGI server used for Django Channels, supporting WebSockets and HTTP2, essential for real-time message updates.

Advantages:

- Handles both HTTP requests and WebSocket connections efficiently.
- Designed to work with Django Channels, enabling asynchronous communication.

7.9. asgiref and Redis: Used for asynchronous support and WebSocket management. Redis is often used for managing channel layers, and ASGI serves as the interface for handling real-time protocols.

Advantages:

- Ensures low-latency, high-throughput messaging.
- Easily handles real-time communication between multiple users.

8. Key Improvement Areas for Chatly

8.1. Scalability Enhancements:

- **Database Upgrade:** Move from SQLite to a more robust and scalable database like PostgreSQL or MySQL for production to handle large user bases and higher traffic.
- **WebSocket Scalability:** Use Redis with Django Channels to handle a large number of WebSocket connections, ensuring smooth real-time updates for multiple users simultaneously.

8.2. File Storage Improvement:

- **Move from Local Storage to Cloud Storage:** Replace local file storage with Amazon S3, Google Cloud Storage, or Cloudinary to ensure secure, scalable, and globally accessible file storage.
- **File Compression:** Implement file compression techniques for file uploads (e.g., images, and videos) to reduce storage and bandwidth usage.

8.3. Security Enhancements:

- **End-to-end Encryption:** Implement end-to-end encryption for chat messages to enhance data privacy and security, ensuring that only the intended recipients can read the messages.
- **Token-Based Authentication:** Move from session-based authentication to JWT (JSON Web Tokens) for API calls, especially when the front end and back end are separated in a production environment.
- **Input Validation and Sanitization:** Add comprehensive validation and sanitization for user inputs, especially in message fields, to prevent XSS, CSRF, and SQL injection attacks.

8.4. Real-Time Features Expansion:

- **Typing Indicators and Read Receipts:** Implement real-time typing indicators and read receipts in the chat application to improve the user experience.
- **WebSocket Optimization:** Reduce unnecessary WebSocket traffic and optimize real-time messaging by implementing techniques like message batching or client-side message throttling.

8.5. Improved User Experience:

- **Mobile Responsiveness:** Ensure full mobile responsiveness and cross-browser compatibility for better accessibility across devices (mobile, tablet, desktop).
- **Offline Functionality:** Implement offline capabilities using technologies like service workers to allow users to view recent messages and send new ones while offline.
- **Chat Search Functionality:** Add a search feature for users to find specific messages or users within a chat or across chats.

8.6. Load Balancing and Deployment:

- **Load Balancing:** Implement load balancing (using Nginx or HAProxy) to distribute incoming WebSocket and HTTP traffic across multiple servers, improving availability and handling peak loads.
- **Containerization:** Use Docker to containerize the application and simplify deployment and scalability by orchestrating with tools like Kubernetes.

8.7. Monitoring and Analytics:

- **Real-time Monitoring:** Integrate monitoring tools like Prometheus and Grafana to track real-time performance metrics, WebSocket connections, user activity, and system health.
- **User Analytics:** Implement user activity analytics to track engagement and interactions within the chat platform, identifying key areas for user growth and retention.

8.8. User Interface Improvements:

- **Dark Mode:** Add a dark mode option to improve user comfort, especially in low-light environments.
- **Customizable Themes:** Allow users to customize the theme and appearance of their chat interface to provide a more personalized experience.

8.9. Group and Private Chat Enhancements:

- **Admin Controls for Group Chats:** Implement additional controls for group chat admins, such as the ability to mute or remove members, or to set chat permissions (e.g., who can post).
- **Message Pinning and Archiving:** Allow users to pin important messages in both group and private chats, and implement message archiving for better organization.

8.10. Integration with Third-Party Services:

- **Social Login Integration:** Extend authentication options by integrating OAuth for social logins via Google, Facebook, or GitHub, improving the user onboarding process.
- **AI-based Chat Features:** Implement AI-driven chatbots or auto-reply features to enhance the user experience, especially for customer service or automated interactions.

8.11. Audio and Video Calling Features:

- **Integration of WebRTC:** Implement WebRTC (Web Real-Time Communication) for peer-to-peer audio and video calling capabilities within private and group chats.
- **UI Enhancements:** Add call initiation buttons in the chat interface, with options for audio and video calls, and ensure responsiveness across devices.

By focusing on these improvements, the Chatly messaging service prototype will be better equipped for production environments, delivering enhanced performance, security, and user experience.