Blockchain Chat App (Detailed architectural overview)

The following is a detailed architectural overview of the chat application as assigned in the Job exercise by Mr. Thomas Kolmans.

User registration and authentication:

Wallet Connection:

- To register with the ChatApp user first need to sign-up to the platform using their wallets.
- The react application can either integrates Metamask's or any other wallet provider's APIs to allow for a secure wallet connection to the application.
- Alternatively, we can use Moralis provider to look for the injected provider in the browser's window object.

Email verification:

- Upon connecting their wallets, users need to fill in their email address in the registration form.
- Upon submission, the node.js server sends a verification email to the provided email address using AWS SES (Simple Email Service).
- The email contains a unique verification link that expires after a certain time period.

User ID generation:

- Once the email is verified the backend node.js server generates a unique user ID associating it with the user's email and wallet address and other details such as their age, gender, etc.
- The script also stores that user ID in PostgreSQL database.

JWT-based authentication:

- When a user wants to sign-in they are prompted to connect their wallet.
- Once the wallet is connected the react application sends a request to the node.js server to authenticate the user.
- The node.js server verifies the user wallet's public key or signature to verify the authenticity of the request.
- If the authenticity of the request is verified the server then generates a JWT (JSON web token) containing user's ID and their wallet address and other information.
- The JWT returned to the react app is secured safely in the local-storage.

Sensitive Data Transmission:

- All sensitive data, including user email addresses and authentication tokens, are transmitted securely over HTTPS to prevent interception by unauthorized parties.

AWS Services:

- **SES**: Used for sending email verification link to the users.
- **Cognito:** Utilized for managing user identities and access control, including user pools and identity pools.
- **KMS (Key Management Service):** Used for managing encryption keys for securing sensitive data at rest.

User Discovery and Chat Functionality:

Interests:

- Once the user successfully logs on to the platform for the first time, they are asked to select their interests.
- The front-end displays a range of interests that a user might prefer (e.g., Sports, Music, Tech) along with check box on each of them to select.

Back-end processing:

- After interests' selection a form is submitted and the front-end application sends a request to the node.js server.
- The server receives those interests along with the unique user ID.
- The server then processes the request and stores the user's interests in the database.

Database:

- Create a new table in the PostgreSQL database to store user interests.
- Each user's interests are stored as a separate entry in the table, linked to their unique user ID.

Data Storage and Retrieval

- Interests data is stored in a JSON format and associated with a user's profile for easy retrieval.
- When a user logs in to the platform, their interests are retrieved from the database and used to personalize their experience (i.e., to show them other users with similar interests).

Search Functionality:

- The frontend application includes a search component where users can search for other users based on shared interests.
- Users can view the profiles of other users they find through the search feature, including details such as interests, gender and age.

Search Query Back-end Processing:

- When the user searches for other users with same interests the front-end sends a request to the node.js server with the specified interests.

- The backend server processes the search query and retrieves relevant user profiles from the database based on the specified criteria.

- Before displaying search results, the backend server verifies that the requesting user is authenticated and authorized to view other users' profiles using the same JWT stored in local-storage.

- Each search result is presented as a profile card containing essential user information, such as user ID and interests.

Chat Initiation:

- Upon finding the person-of-interest, the user can initiate a chat through the front-end.

- The frontend application sends a chat initiation request to the backend server, indicating the sender and recipient users' IDs.

- The recipient receives a real-time notification implemented using “Socket.io” indicating that they have received a chat request from the sender user.

- The recipient can either accept or decline the request. If they accept a new chat is initiated.

Socket.io Integration:

- Socket.io is integrated to the node.js server to establish and manage real-time connections of connected clients (i.e., connected web browsers).

- When an event occurs that necessitates sending a real-time notification to one or more clients (such as receiving a chat message or a chat initiation request), the server emits the notification to the respective clients via Socket.io.

- Upon loading, the frontend application establishes a Socket.io connection with the backend server.

Chat Functionality

- Upon user authentication and connection establishment, the backend server assigns the user to specific Socket.io rooms based on the user's context, such as the chat rooms they are part of or the conversations they are engaged in.

- Once the connection is established, the frontend application listens for incoming notifications emitted by the server.

- When an event occurs that requires sending real-time notifications to users within a specific context (e.g., receiving a chat message in a particular chat room), the server emits the notification to all users within the corresponding Socket.io room.

- When a notification or event is received, the frontend application updates the user interface accordingly to reflect the new information specific to the context of the Socket.io room.

Benefits of using socket.io rooms for chat functionality:

- **Scalability:** Socket.io rooms facilitate scalable real-time communication by allowing for targeted event emission to specific groups of users.

- **Organization:** Rooms provide a structured way to manage and organize users based on their context, such as chat rooms or conversations, improving the efficiency of real-time communication.

- **Customization:** Rooms offer flexibility in managing real-time notifications and events to the specific needs and requirements of different user groups or contexts within the chat application.

Data Encryption and Decryption for Chat Privacy:

To ensure the privacy and security of chats between users, encryption and decryption mechanisms can be implemented:

Encryption Process:

- When a user sends a message, the frontend application encrypts the message using a symmetric encryption algorithm (e.g., AES).

- The encryption key used for encryption is generated dynamically for each chat session.

- Additionally, the frontend application signs the encrypted message using the sender's private key to ensure message integrity and authenticity.

- Once encrypted and signed, the message is sent to the backend server for storage and transmission to the recipient.

Backend Processing:

- Upon receiving the encrypted message, the backend server securely stores the message data in the database.

- The server does not have access to the encryption key or the plaintext message, ensuring data privacy even if the server is compromised.

Decryption Process:

- When a user retrieves chat messages from the database or receives a new message, the frontend application fetches the encrypted message from the server.

- The frontend application decrypts the message using the corresponding decryption key, which is securely exchanged between the sender and recipient.

- After decryption, the frontend application verifies the message signature using the sender's public key to ensure message integrity and authenticity.

- The decrypted message is then displayed to the recipient user in the chat interface.

Key Exchange:

- To securely exchange encryption keys between users, asymmetric encryption algorithms (e.g., RSA) can be used.

- When users initiate a chat session, they can exchange public keys securely using Diffie-Hellman key exchange protocol.

- Once both parties have exchanged public keys, they can securely derive a shared encryption key for symmetric encryption of chat messages.

Data Privacy:

- Encrypting chat messages ensures that even if the message data is intercepted or accessed by unauthorized parties, it remains encrypted and unreadable without the decryption key.

- This provides end-to-end encryption of chat messages, preserving the privacy and confidentiality of user conversations.

Crypto Tipping system:

UI Component:

- A UI component in the front-tend application allows users to enter the amount of ETH they want to tip and trigger the tipping process.
- This component should include an input field for the tip amount, a "Send" button to initiate the tip, and any necessary UI elements to display transaction status or errors.
- Implement event handlers for user interactions, such as clicking the "Send" button.
- When the user triggers the tipping process, initiate an API request to the back-end server to start the transaction.

Back-end Implementation:

- An API endpoint on the node.js server will handle tip requests from the frontend.
- This endpoint will receive the tip amount and any other necessary data from the frontend and initiate the tipping process.
- The tip amount will be validated to ensure that the user has sufficient funds in their wallet to cover the tip.
- The transaction data, including the recipient's Ethereum address and the amount of ETH will be sent.
- The transaction data (e.g., recipient address, tip amount) will also be sent back to the frontend in the API response.
- Ensure that the response includes any necessary information required by the frontend to interact with MetaMask and confirm the transaction.

MetaMask Integration:

- In the front-end application, MetaMask's JavaScript API will trigger the MetaMask pop-up window when the user initiates the tipping process.
- The transaction data received from the back-end will be passed to MetaMask for user confirmation.

- Once the user confirms the transaction within the MetaMask pop-up window, MetaMask will sign the transaction using John's private key.

- MetaMask will return the signed transaction data to the frontend application.

Transaction Submission:

- The use of web3.js or a similar Ethereum JavaScript library in the frontend application will be used to submit the signed transaction to the Ethereum network.

- The recipient address is specified and the amount of ETH to send in the transaction data.

- Upon receiving the signed transaction from MetaMask, submit the transaction to the Ethereum network using Web3.js's `sendSignedTransaction` method.

Transaction Confirmation:

- The submitted transaction is monitored on the Ethereum network for confirmation.

- Once the transaction is confirmed, The UI in the front-end is updated to notify the user that the tip transaction to Jane was successful.

Required Tech Stack:

- **Front-end**: React for building the user interface components.
- **Back-end**: Node.js back-end server hosted on AWS or Azure.
- **Database**: PostgreSQL for storing user data, including interests.
- **Storage**: AWS S3 or Azure Blob Storage for storing user profile pictures and other media files.
- **Authentication**: MetaMask wallet integration for user authentication.
- **Real-Time Communication**: Socket.io for real-time messaging.
- **Blockchain Integration**: Ethereum blockchain for the tipping feature.
- **Blockchain Interaction**: web3.js or ethers.js libraries for blockchain data interaction.
- **Site Recovery**: We may need to implement AWS backup for site recovery.
- **DDoS Mitigation**: AWS shield for DDoS attacks.
- **Server latency and server load**: AWS Cloudfront or Cloudflare CDN for faster content delivery.

Scalability Solutions for Handling Increased User and Message Loads:

Auto-Scaling:

- Configuration of auto-scaling policies to automatically add or remove server instances based on predefined criteria such as CPU utilization, memory usage, or network traffic.
- AWS auto scaling can adjust the number of these instances dynamically.

Optimized resource utilization:

Consistent monitoring of resource utilization metrics such as CPU, memory, and network bandwidth to identify bottlenecks and optimize resource allocation.

AWS Elastic Load Balancer (ELB) to evenly distribute traffic and prevent overload on any single server instance.