System Design Document: Messaging Service

Mayank Patel(IIT Bhilai)

Contents

1 Introduction

This document provides the system design for the Messaging Service Prototype. System enables users to register, **authenticate, send and receive messages, participate in group chats, and experience real-time updates**.

The system leverages real-time communication to ensure instant message delivery, enhancing user experience. Built with modern web technologies like **Next.js** for the frontend and **Node.js with Express.js, NextJs** for the backend, the system offers a scalable, efficient architecture. A NoSQL database, **MongoDB**, is used to manage user data and messages, while **Socket.io** facilitates real-time updates.

2 System Architecture

High-Level Architecture(Design)

In this chat web application, client will be on web application. Each client connects to the chat services. Based on this chat service has following feature:

- In web application, client can receive message from other client.
- It pass the message to the client which is connect to it based on the group which they have joined
- If the client is not online, the server has to hold the message for the duration until they are not online

With the help of HTTP protocols, it is difficult to achieve all this feature because HTTP is client initiated, it cannot send message from server to client.
Techniques which are server-initiated are:

1. **Polling:** Client periodically request data from the server(inefficent due to high number of unrequired request).
2. **WebSocket:** Bidirectional communication protocol that enable time communication between the client and server over a single, long-lived connection.

Hence, **WebSocket is preferred over HTTP(S) protocol**, since HTTP does not keep the connection open for the longer period of time for the server to send data to a client.
We can implement web-socket with the help of **Socket.io** npm package which upgrade the upgrade header value of the HTTP request to web socket.

Step-by-Step complete communication between client and the server:

1. User-1 and User-2 create a communication channel with the chat server.
2. User-1 send the message to the chat server.
3. When a message is received, the chat server acknowledges back to user A.
4. Chat server sends the message to user B and stores the message in the database if the receiver's status is offline.
5. User B sends an acknowledgment to the chat server.
6. Chat server notifies user A that the message has been successfully delivered.
7. When user B reads the message, the application notifies the chat server.
8. Chat server notifies user A that user B has read the message.

There are three major categories in the messaging-prototypes applications:

1. **Stateless Services:** Manages login, sign-up and user profile.
2. **Stateful Services:** Chat Service
3. **Third-party integration:** For push-notication to message arrival

Chat and User data storage

There are two types of data in typical chat system:

- Generic Data: User Information(name, email, phone, password).
- Chat History Data: Store the history of the chats and it information(arrival time,created time).

Key-value storing will be best for this data models because it provide low latency(minimum delay) to access data and also have easy horizontal scaling. We can implement the key-value storing with the help of non-SQL database **Mongo-DB**.

Data Models

For one-on-one messages:
Two columns: message_id(id of the message), created_at(when rhe message is created by user)
Primary Key: message_id(because it is unique)

For group messages:

Two columns: group_id(id of the group) ,message_id(id of the message), created_at(when the message is created by user)
Primary Key: union of (group_id ,message_id
)

Message ID is responsible for the order of the messages.
With the help of **MongoDB aggregation pipeline** we can recreate this model data.

!!! For Large-Scale Model !!!

- For the large-scale model, one Web-Socket server will not be enought to handle millions or billions of users, there is a **need of multiple Web-Socket Server** and the manager which can manage multiple server.
- WebSocket server also need to communicate to another message service
- WebSocket servers **don't keep track of groups** because they only track active users. So, there should be group message handler and group message services.

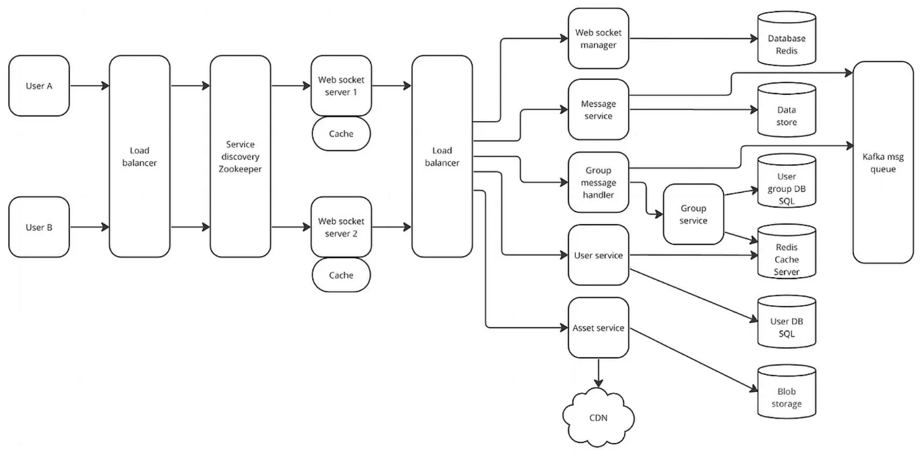


Figure 1: High-Level and Largely Scalable Messaging Model

3 My Project Design(Github Link)

3.1 Website Flow(Structure):

- New user should first sign-up and create there account(the detail of the user will be stored in database),
- After creating account, login in with your credential, to obtain the access of the website.
- User will redirect to the home page which.
- Now, user can join the group, **if he/she know the group code**.
- **Else** , need to create one group with the name and code(which will be store in the database).
- After creating the room enter the code
- You will enter the room where you can send messages to and receive message from another users.

USER AUTHENTICATION
Library: NextAuth.js (open-source for Next.js)
Built-in support for many popular services

Custom Credential:

- **Sign-up/Login**: Uses Email or Username as identifier
- Password-based authentication
- **Returns**:
  - JWT Token
  - Session with ID, Username, Group Code

Session Management:

- Handled via *SessionProvider*
- Keeps user session updated and synced across tabs

SENDING AND RECEIVING MESSAGES

With the help of APIs(Sending and Receiving)

- Created a APIs for getting, sending and receiving messages

REAL-TIME MESSAGE UPDATES

With the help of Socket.io

- A socket.io server is created within the application.
- Server listen for websocket connection and setup the event handler for different type of messages.
- When a client (e.g., a web browser) connects to the server, a persistent WebSocket connection is established. The client can now emit events to the server.
- The client sends messages to the server using the emit function.
- The server listens for messages from clients. When a message is received, it can be processed and then broadcasted to other clients.

GROUP CHAT FUNCTIONALITY

With the help of Socket.io

- Using the socket function, **socket.broadcast.emit** server sends the receive message to all connected clients except the one that sends it.

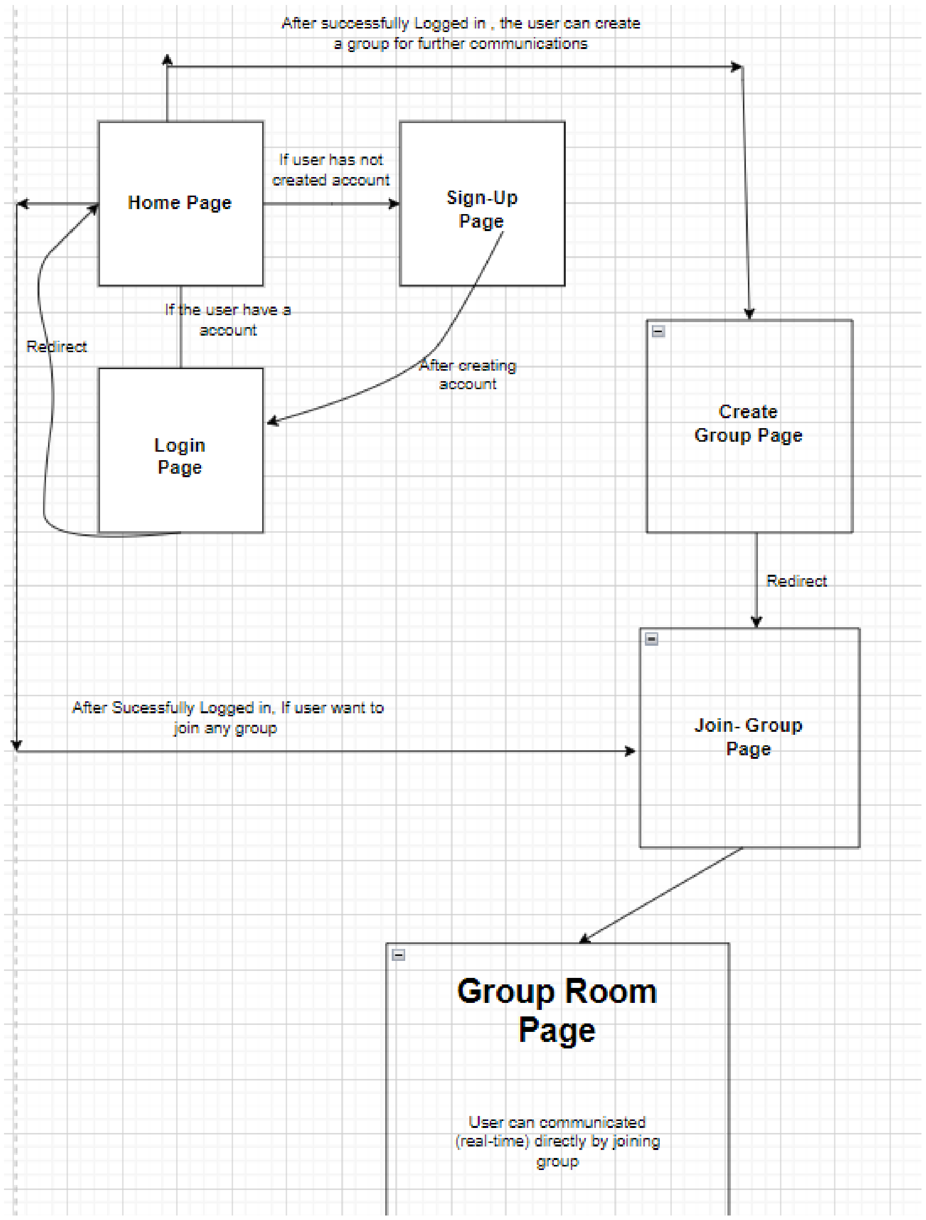

After successfully Logged in , the user can create a group for further communications
Home Page
If user has not created account
Sign-Up Page
If the user have a account
Redirect
After creating account
Login Page
Create Group Page
Redirect
After Sucessfully Logged in, If user want to join any group
Join- Group Page
Group Room Page
User can communicated (real-time) directly by joining group

Figure 2: Website Flow diagram

3.2 Databases:

I have use the mongoDB database to store information(named: chatapp).
Two types of collection:

1. **Users(To store user information):** It contains

- username(type: String)
- email(type: String)
- password(type: String)
- status(type: Boolean)
- messages(type: Array) - Stores the user messages

2. **Groups(To store group information):** It contains

- groupname(type: String)
- code(type: String) - For joining
- members(type: 'User')
- createdBy(type: 'User') - User who created the group

We can add messages(in group) to ensure the old user messages of the user are been stored there.

3.3 Project File Structure:

```
    // Root Directory
+-- app
    +-- (auth)  -> Contains the pages
    |   +-- about
        |   +-- page.jsx
        +-- chat
        |   +-- [room]
            |   +-- page.jsx
        +-- create-room
        |   +-- page.jsx
        +-- join-room
        |   +-- page.jsx
        +-- login
        |   +-- page.jsx
        +-- sign-up
        |   +-- page.jsx
    +-- api   -> Contains the api
        +-- auth
        |   +-- [...nextauth]
            |   +-- option.js
            |   +-- route.js
        +-- get-message
        |   +-- route.js
        +-- send-message
        |   +-- route.js
        +-- suggest-message
        |   +-- route.js
        +-- accept-message
        |   +-- route.js
        +-- sign-up
        |   +-- route.js
        +-- group-join
        |   +-- route.js
        +-- group-creation
        |   +-- route.js
        +-- socket
        |   +-- route.js
    +-- components -> Contains the component used in the pages
    |   +-- Block.jsx
    |   +-- Box.jsx
    |   +-- Footer.jsx
    |   +-- Navbar.jsx
    +-- global.css
    +-- layout.js
    +-- not-found.js
    +-- page.js
+-- context -> To maintain the global state and application level data
|   +-- AuthProvider.js
+-- model -> Contains the model for the databases
|   +-- Group.js
|   +-- Message.js
|   +-- User.js
+-- utils
    +-- dbConnect.js
+-- .env
+-- jsconfig.json
+-- middleware.js (To Handle the middlewares)
+-- next.config.mjs
+-- package-lock.json
+-- package.json
+-- server.js
+-- tailwind.config.js
+-- README.md
```

3.4 AI Integration:

In this Day-Day world, AI(Artificial Intelligence) is overtaking other technology.
How I implement AI in the chat application?

- Using **Vercel AI SDK toolkit** to built AI-Integration in the application.
- With **OpenAI Provider** it enables integration with the OpenAI models and APIs
- **User will click on Suggest message, then Prompt will go to Ôpen AI, it will respond to that prompt and give the output to the user.**

3.5 Addition Features:

- **Logout Option:** For the user who wants to exit.
- **No-Found Page:** If the user access any other page which is not present it will be redirected to the No-Found Page.
- **Message Date:** Shows the date of the message send.

3.6 Solution(Optional Feature):

Video and audio calling feature:
To add the video and audio calling feature for the messaging app, it can be done with the help of **WebRTC(Web Real-Time Communication** for person to person audio/video communication, and with socket.io to handle the signaling process for establishing WebRTC connection.

- Before WebRTC connection, user needs to exchange information about, How to connect.
- Socket.io can be used as the signaling server to exchange ICE candidates, SDP (Session Description Protocol) offers, and answers.
- Add the UI and the button for audio and video calls.
- Ensuring the access of the video and microphone permissions.

Steps to establish the WebRTC:

1. **Get User Media**: Capture the media (audio/video) from the user's device using `getUserMedia()`.
2. **Signaling**: Use a signaling server (e.g., using `socket.io`) to exchange messages between the peers to establish a WebRTC connection.

3. **Establish Peer Connection**: Use WebRTC's `RTCPeerConnection` API to set up the connection between peers.
4. **Exchange SDP offers/answers**: One peer sends an offer, and the other peer responds with an answer. These contain the media session description using SDP (Session Description Protocol).
5. **Exchange ICE Candidates**: ICE (Interactive Connectivity Establishment) candidates are network information that peers exchange to establish connectivity.
6. **Connect Media Streams**: Once the connection is established, the media streams (audio/video) are sent directly between the peers.

4 API Design

4.1 Authentication API

Authentication API is responsible for managing user registration and login functionalities. It allows users to securely register and authenticate to the messaging service. NextAuth.js is used in combination with custom credentials to handle the authentication process.

- **POST /api/sign-up**: This API is used to register a new user in the system. When a user submits their email and password, the backend stores the user details in the MongoDB database after performing necessary validations (e.g., checking if the email is already registered and checking if the useraname already exist or not). Passwords are typically hashed using bcrypt/bcryptjs which the method of salting.

Key points:

- **Custom Credentials with NextAuth.js**: NextAuth.js provides an out-of-the-box solution for authentication, and by using a custom credentials provider, we can define our own logic to authenticate users based on email and password.
- **JWT for Stateless Authentication**: The system uses JWT for stateless session management, ensuring that user sessions are secure and scalable. The token is stored on the client side (usually in cookies or localStorage) and sent with each request to authenticate the user.
- **Security Considerations**: Passwords are securely hashed using bcrypt, and JWT tokens are signed with a secret key. The server never stores plain-text passwords, reducing the risk of leaks.

The Authentication API is built with scalability and security in mind, ensuring a smooth user experience while protecting sensitive user data.

4.2 Messaging API

Accept-Message:

This API is used to create connection for the user whether he/she is online or offline which means whether it can receive message or not.

Pseudocode:

```
Function acceptMessage POST{
  if(db.Connected)
      {
          //Get the user from the session if there
             is session
          if(noSession || noSessionUser)
              {
                  return Not an authenticated user
              }
          //Extract ID from the user and fetch the
             connection status from the request
          //Update the user connection status and
             Update the user
          if(user.isNotUpdate)
              {
                  return Fail to update the
                     userstatus
              }
              return Message acceptance status
                 updated successfully
      }
    Else
        {
            return Not Authenticated
        }
}
```

Send-Message:

This API is used to send a text-message from the sender to the receiver by making POST API call. It is created by taking session user and ID.

Pseudocode:

```
Function sendMessage POST(req)
{
    if(db.connected)
        {
            // Get the username and message from
                the request
            if(user.notpresentInDB)
                {
                    return User not found
                }
            // If user is not accepting the
               message
            if(user.isOffline)
                {
                    return User not accepting
                       the message
                }
            // Create new message and push it
               into user's collection
            return Response(Message sent
               successfully)
        }
    else
        {
            return Error in adding message
        }
    }
```

Get-Message:

This API is used to fetch the user to read all the unread messages when the connection status is been changed.

Pseudocode:

```
    Function getMessage GET{
  if(db.Connected)
      {
          //Get the user from the session if there
             is session
          if(noSession || noSessionUser)
              {
                  return Not an authenticated user
              }
          //Get the ID of the user and change it to
             string for aggregation
          //Aggregate the userData[match with id,
             unwind the message column, sort
             message based on when they are created
             , group them all]
          //After aggregation
          if(user.notFound || user.length==0)
              {
                  return User not Found
              }
          Else
              {
                  return messages
              }
      }
   Else
       {
           return Internal Server Error
       }
}
```

Delete-Message:

This API is used to delete the already existing message.

Pseudocode:

```
Function DELETE() {
if(db.Connected)
{
    //Extract the user ID from the parameter
    //Get the DB Connection
    //Get the current user-session and Extract
       user from int
    if(user.isNotPresent || session.isNotPresent
       )
        {
            return "Not Authenticated"
        }
    //Updating the user messages to remove the
       message br it ID

    if(resultUser.notModified)
        {
            return Message not found
        }
    return Message Deleted Successfully
    }
    if(AnyError) return Generic Error/Internal
       Server Error
}
```

Suggest-Message:

This API is use for message suggestion, where user get the suggestion from the OpenAI model based on there messages or prompt.
It takes in a request, generates suggestions for open-ended questions, and returns them as a response in a streaming format for better performance. The generated questions are suitable for social messaging platforms and are designed to be engaging and universally appealing.

4.3 Group-Chat API

Group-creation

This API will create a group chat-room.

Pseudocode:

```
Function POST(req)
{
    if(db.isConnected)
        {
            //Fetch the groupname and code
            //check the groupname or code are
               unique or Not
            //Get the session and extract the
               user from the session
            if(!session || !user)
                {
                    return Logged in first
                }
            //Store the data in the database
            return Group created sucessfully
        }
    else{
        return Error in creating group
    }
}
```

Join-GroupChatRoom:

This API is for the user to join the group chat room.

Pseudocode:

```
Function POST(req)
 {
     if(db.isConnected)
         {
             //Fetch the code
             if(code.notPresent)
                 {
                     return No code is present
                 }
             //Get the session and extract the
                user from the session
             if(!session || !user)
                 {
                     return Logged in first
                 }
             //Get the user if from the user
                session
             //Get the group from the code
             if(group.NotFound)
                 {
                     return Group NotFound
                 }
             //Add the user to the group members
                in the group dataset
             //Store the data in the database
             return Successfully join the group
         }
     else{
         return Error in joining group
     }
 }
```

We can create a another API for to retrieve the group chat details with the help of group id.

5 Technology Choices and Justification

Dependencies and Libraries

The following dependencies and libraries are used in this project:

- **axios**: A promise-based HTTP client for the browser and Node.js, used for making API calls.
- **bcrypt** & **bcryptjs**: Used to securely hash passwords for authentication and storage.
- **cors**: Middleware that allows for Cross-Origin Resource Sharing, ensuring that the API can be accessed from other domains.
- **http**: Provides basic server utilities (part of Node.js core library).
- **mongoose**: A MongoDB object modeling tool that allows interaction with MongoDB for storing user data and messages.
- **next**: A React-based framework for building server-side rendered applications.
- **next-auth**: Handles authentication for Next.js applications with multiple providers (e.g., email, OAuth).
- **nodemon**: Used for automatically restarting the server during development when code changes are made.
- **react** & **react-dom**: Core libraries for building user interfaces in React.
- **react-hook-form**: Provides a simple way to handle form input and validation in React applications.
- **react-router-dom**: Helps manage routing in React, allowing for navigation between different parts of the app.
- **socket.io**: Enables real-time, bi-directional communication between the client and server for group messaging.
- **zod**: A schema validation library used in conjunction with forms to validate input.
- **tailwindcss**: A utility-first CSS framework used for styling the user interface.

DevDependencies

- **autoprefixer**: Automatically adds vendor prefixes to CSS, ensuring cross-browser compatibility.
- **eslint** & **eslint-config-next**: Linting tools to maintain code quality and enforce coding standards.
- **postcss**: A tool for transforming CSS with JavaScript plugins.
- **tailwindcss**: Extends Tailwind's capabilities for styling the frontend.

Why These Technologies Were Used

- **Next.js**: Chosen for its server-side rendering capabilities and ease of building scalable React applications. It also allows for fast development and optimized production builds.
- **Socket.io**: Used to handle real-time group chat communication, ensuring users receive messages instantly.
- **MongoDB & Mongoose**: MongoDB is a NoSQL database that stores user data and chat messages, and Mongoose helps model the data and interact with MongoDB more effectively.
- **Tailwind CSS**: Tailwind makes styling fast and consistent by allowing us to use utility classes directly in our JSX.
- **Next-Auth**: Simplifies the authentication process by providing out-of-the-box solutions for session handling and OAuth integration.

6 Setup Instructions

6.1 Clone the Repository

To set up the project locally, clone the repository:

```
git clone https://github.com/MayANKPaTeL2303/Group-
    Messenger-Messaging-Service-Prototype.git
cd Group-Messenger-Messaging-Service-Prototype
```

6.2 Install Dependencies and packages

Install the required dependencies and packages:

```
npm install axios bcrypt bcryptjs cors http mongoose
    next next-auth nodemon react react-dom react-hook-
    form react-router-dom socket.io zod tailwindcss
    eslint autoprefixer
```

6.3 Run the server

Run the server(for development) - (localhost: 3000)

```
npm run dev
```

Run the server(for production)

```
npm run build
```

6.4 For Chatting,

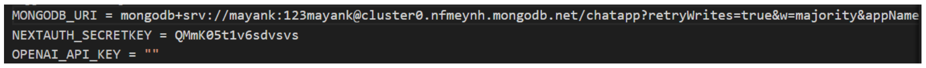
Run the another terminal(localhost:3001) and repeat the process of sign-up and login.

```
npm run dev

nodemon server.js
```

6.5 Environment Variables

The .env file stores the environmental variable:



```
MONGODB_URI = mongodb+srv://mayank:123mayank@cluster0.nfmeynh.mongodb.net/chatapp?retryWrites=true&w=majority&appName
NEXTAUTH_SECRETKEY = QMmK05t1v6sdvsvs
OPENAI_API_KEY = ""
```

Figure 3: Environmental variables

7 Conclusion

This system design outlines the architecture and flow of the messaging service prototype. With a modular frontend, scalable backend, and a real-time message update system, the application is optimized for performance and usability.