VIETNAMESE GERMAN UNIVERISTY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEER

Distributed System Module

REALTIME CHAT APPLICATION PROJECT

Instructor: Dr. Phan Trong Nhan

Student 1:	Nguyen Dai Minh	10421037
Student 2:	Nguyen Khoi Nguyen	10421097
Student 3:	Luu Minh Khang	10421042
Student 4:	Nguyen Hoang Nguyen	10421096
Student 5:	Huynh Dang Trinh	10421064

June 13, 2024

Content

1 Introduction

The primary objective of this project is to develop an application where multiple users can communicate with others in real-time. This includes the ability to register, log in and log out with unique credentials. Also some fault tolerances including chat message storage when client disconnect as well as automatic failover mechanism that we developed to enhance the experience of our users.

2 Functional Requirements

The functional requirements outline the essential features that the application must support. These are the core functionalities necessary for the application to operate effectively.

2.1 Connect to the database

The server and back-up server first have to connect to the database (in this project we use postgres) to store data (users' name, passwords, chatting data).

```
1    //Connect to database (postgres)
2    const pool = new Pool({
3      user: 'postgres',
4      host: 'localhost',
5      database: 'chat',
6      password: '********',
7      port: 5432,
8    });
```

Connect server to the database

- Creates a new connection pool to a PostgreSQL database using the `Pool` class from the `pg` module.
- The `Pool` constructor is called with an object that contains the configuration for the database connection, including:
  - Username: `'postgres'`
  - Host: `'localhost'`
  - Database name: `'chat'`
  - Password: `'********'`
  - Port: `5432`
- The connection pool is stored in the `pool` constant, which can be used to query the database.

2.2 Read messages and users' data from database

```
//Read messages history from database
pool.query('SELECT * FROM chat_history ORDER BY id', (error, results) => {
    if (error) {
      throw error;
    }
    chatStore = results.rows.map(row => ({msg: row.content, user: row.
        username}));
  });

  //Read user information from database
pool.query('SELECT * FROM user_table', (error, results) => {
    if (error) {
      throw error;
    }
    results.rows.forEach(row => {
      const username = row.username;
      const password = row.password;
      if (username && password) {
        userStore[username] = { password };
      }
    });
  });
```

Read messages and users' data from database

- **Query 1: Retrieve Chat History**
  - Queries the database to retrieve all records from the `chat_history` table, ordered by `id`.
  - If there's an error during the query, it throws an error.
  - If the query is successful, it maps through the results and stores each row as an object with `msg` (message content) and `user` (username) properties in the `chatStore` array.
- **Query 2: Retrieve User Information**
  - Queries the database to retrieve all records from the `user_table`.
  - If there's an error during the query, it throws an error.
  - If the query is successful, it iterates through the results.
    - For each row, if both `username` and `password` exist, it adds an object to the `userStore` object with the `username` as the key and the `password` as a property of the object.
  - Inside the server also has this query so that each time it moves to the back-up server, the server still have information of the previous registers in the main server.

2.3 User Registration, Login and Log out

The application must allow users to create new accounts by providing the necessary information such as a username and password. After registration, users will be able to log in using their credentials. Successful login should grant access to the chat interface. After finishing the conversation, users can log out of their accounts to ensure security or to change the account.

Our codes and graphical interface for this requirement:

```
// Register User
socket.on('register', function(data, callback){
    if(userStore[data.username]){
        callback({ success: false, message: 'Username already taken' });
    } else {
        bcrypt.hash(data.password, 10, async function(err, hash) {
            if(err) {
                callback({ success: false, message: 'Error registering user'
                    });
            } else {
                userStore[data.username] = { password: hash };
                await pool.query('INSERT INTO user_table (username, password
                    ) VALUES ($1, $2)', [data.username, hash]);
                callback({ success: true });
            }
        });
    }
});
```

Registration function of users

This code snippet manages user registration by:

- Checking username availability.
- Securing the password using hashing.
- Storing the user credentials.
- The 'pool.query' is used to insert user names and the hash of the passwords to the database
- Providing appropriate feedback on success or failure.

```
// Login User
socket.on('login', function(data, callback){
    if(!userStore[data.username]){
        callback({ success: false, message: 'Invalid username or password'
            });
    } else {
        bcrypt.compare(data.password, userStore[data.username].password,
            function(err, res) {
            if(res) {
                socket.username = data.username;
```

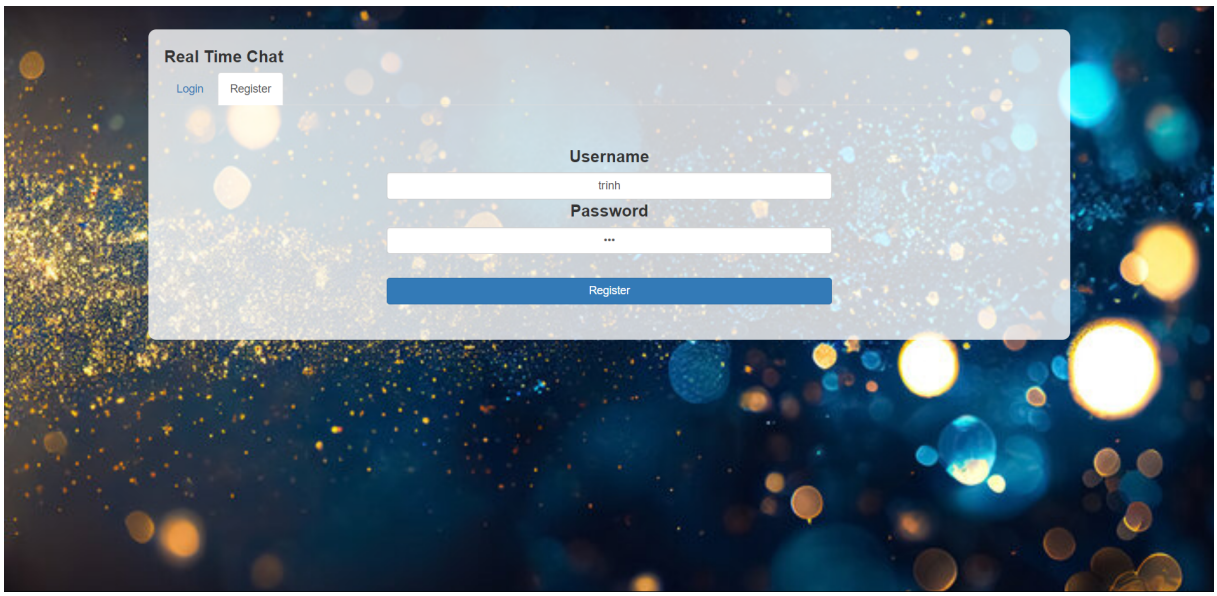

Figure 1: Register screen

```
                    users.push(socket.username);
                    updateUsernames();
                    callback({ success: true });
                } else {
                    callback({ success: false, message: 'Invalid username or
                        password' });
                }
            });
        }
    });
```

Log in function of users

This code snippet manages user login by:

- Checking if the username exists in the 'userStore'.
- Verifying the provided password against the stored hash using 'bcrypt'.
- Handling successful logins by updating the socket with the username and adding the user to the list of connected users.
- Providing appropriate feedback on success or failure through a callback function.

```
    // Logout User
    socket.on('logout', function(callback){
        if(socket.username){
            users.splice(users.indexOf(socket.username), 1);
            updateUsernames();
            socket.username = null;
            callback({ success: true });
        } else {
```

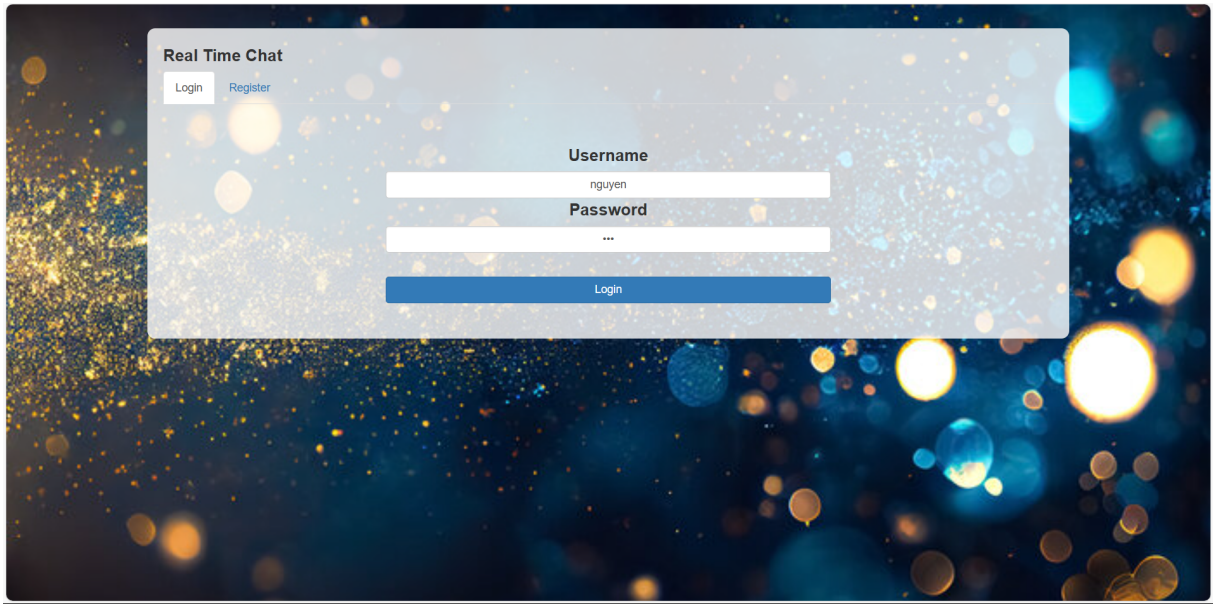

Figure 2: Login screen

```
            callback({ success: false, message: 'No user is currently logged in.
                ' });
        }
    });
```

Log out function of users

This code snippet handles user logout by:

- Checking if a user is currently logged in.
- Removing the user from the active users list.
- Updating the client-side with the new list of active users.
- Clearing the user's session on the server side.
- Providing appropriate feedback on success or failure through a callback function.

```
    function updateUsernames(){
        io.sockets.emit('get users', users);
    }
```

updateUsernames() function

The logout button can be seen in the chatting screen:

2.4 Real-time Messaging

Users should be able to send and receive messages in real-time. This involves:

- **Multiple-User Chat:** Multiple users can chat to others at the same time.

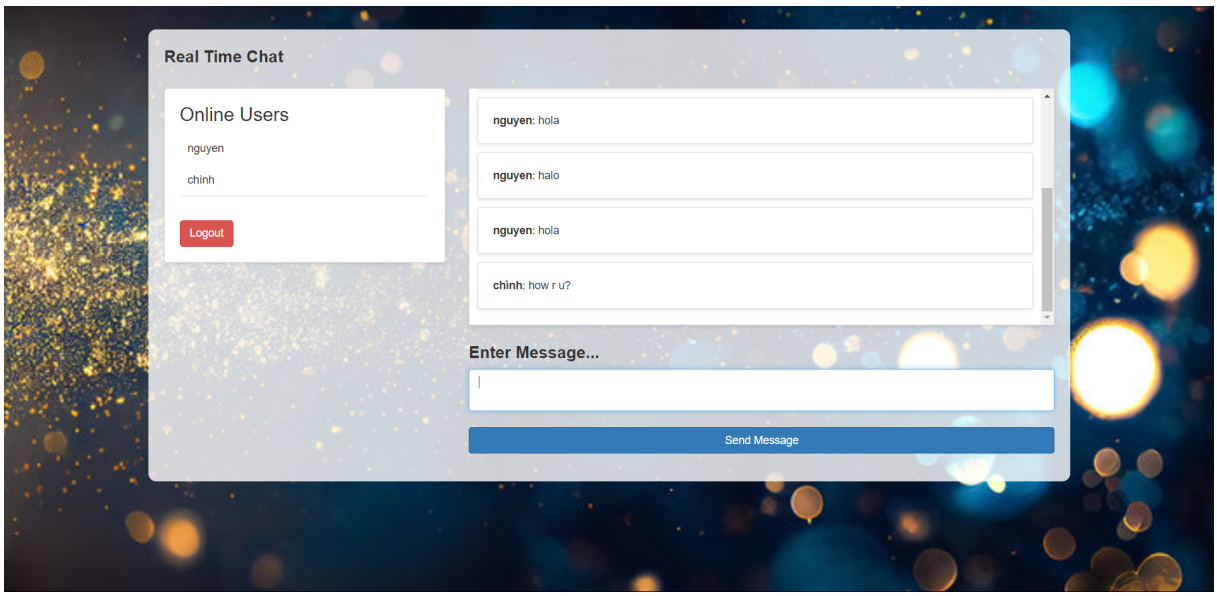

Figure 3: Chatting screen with log out button

- **Instant Delivery:** Messages should be delivered instantly, with minimal latency, ensuring a seamless communication experience.

```
  // Send Message
  socket.on('send message', async function(data){
    if (!socket.username) {
        socket.emit('reload');
    } else {
        var message = {msg: data, user: socket.username};
        chatStore.push(message);
        io.sockets.emit('new message', message);
        await pool.query('INSERT INTO chat_history(content,username) VALUES (
            $1, $2)', [data,socket.username]);
    }
});
```

Chatting between users

- **Event Listener**
  - Listens for 'send message' event on the socket connection, which is emitted from the client-side when a user sends a message.
- **Asynchronous Function**
  - Triggered when the 'send message' event is received, taking 'data' (message content) as a parameter.
- **Username Check**
  - If `socket.username` is falsy (user not logged in), emits 'reload' event back to the client to trigger a page reload.

  - If `socket.username` is truthy (user logged in), proceeds with message processing.
  - This is mainly used for the back-up server to automatically reload the page when the main server is collapsed.

- **Create Message Object**
  - Constructs a 'message' object with `msg` (message content) and `user` (username) properties.
- **Store in Chat**
  - Pushes the 'message' object to the `chatStore` array, which holds messages for the current chat session.
- **Broadcast Message**
  - Emits 'new message' event to all connected sockets, updating all users in the chat with the new message.
- **Database Insertion**
  - Inserts the message into the database using a SQL query.
  - Uses `pool.query` to execute the query, inserting message content and username into the `chat_history` table.
  - Awaits the query completion as it is an asynchronous operation.

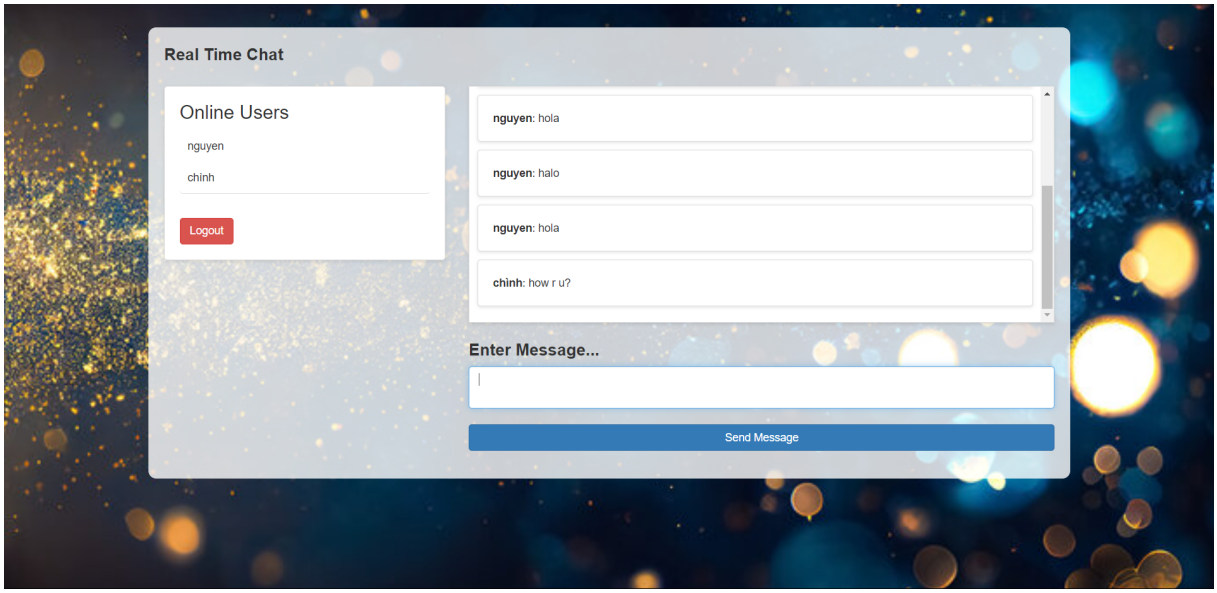

Figure 4: Chatting screen

```
1    io.sockets.on('connection', function(socket){
2    // Connect Socket
3    connections.push(socket);
```

```
4     console.log('Connected: %s sockets connected', connections.length);
5     socket.emit('load all messages', chatStore);
6
7     // Disconnect
8     socket.on('disconnect', function(data){
9         if(socket.username){
10            users.splice(users.indexOf(socket.username), 1);
11            updateUsernames();
12        }
13        connections.splice(connections.indexOf(socket), 1);
14        console.log('Disconnected: %s sockets connected', connections.length);
15    });
```

Process of connect and disconnect to socket

- **Add Socket to Connections**
  - `connections.push(socket);` adds the newly connected socket to the `connections` array to keep track of all active connections.
- **Log Number of Connections**
  - `console.log('Connected: %s sockets connected', connections.length);` logs a message to the console indicating the number of currently connected sockets using the length of the `connections` array.
- **Send Chat History to Client**
  - `socket.emit('load all messages', chatStore);` sends an event named 'load all messages' to the newly connected client, along with the `chatStore` array, which contains the chat history. The client can handle this event to load the chat history.
- **Setup Disconnect Event Listener**
  - `socket.on('disconnect', function(data){...});` sets up an event listener for when the socket disconnects. This function executes when the 'disconnect' event is fired.
- **Handle Disconnect**
  - Checks if the socket has a username with `if(socket.username){...}`.
  - If the socket has a username, it removes that username from the `users` array and calls `updateUsernames()` to update the list of usernames in the application.
  - Removes the disconnected socket from the `connections` array with `connections.splice(connectior 1);`.
  - Logs a message to the console indicating the number of currently connected sockets with `console.log('Disconnected: %s sockets connected', connections.length);`.

2.5 Server Data Management

The server is responsible for handling all data-related tasks, including:

- **User Information:** Storing registration details such as passwords in hash and usernames.
- **Chat History:** Maintaining a cache of all messages exchanged in the system, organized by user or group.
- **Data Integrity:** Ensuring that all data is stored securely and can be retrieved efficiently when needed.

3 Non-functional Requirements

The non-functional requirements address the quality attributes of the system, such as reliability, performance, and usability.

3.1 Automatic failover

A backup server is set up to periodically replicate the data from the main server. In the event of a main server failure, the backup server can take over, minimizing downtime and data loss.

```
events {
    worker_connections 1024;
}

http {
    upstream backend {
        server <your_IP_address>:3000;
        server <your_IP_address>:3001 backup;
    }
    server {
        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```

NGINX Configuration for Automatic Failover

- `events {}`
  - `worker_connections 1024;`
    - This line sets the maximum number of simultaneous connections that each Nginx worker process can handle. It is set to 1024 connections.
- `http {}`

- upstream backend {}
  - server <your_IP_address>:3000;
    - Specifies the primary server at IP <your_IP_address on port 3000.
  - server <your_IP_address:3001 backup;
    - Specifies a backup server at IP <your_IP_address on port 3001. This server will only be used if the primary server is down or unreachable.
- server {}
  - listen 80;
    - Configures the server to listen for HTTP requests on port 80.
  - location / {}
    - proxy_pass http://backend;
    - Forwards incoming requests to the backend upstream, which balances the load between the primary and backup servers, ensuring continuity in case of a server failure.

3.2 Scalability

Implementing a test when 450 users log into the server at one time:

```
import { connect } from 'socket.io-client';
import { assert } from 'chai';

describe('Chat Scalability', function() {
    this.timeout(30000); // Increase timeout to 10 seconds

    it('should handle 450 users', function(done) {
        let users = Array.from({length: 450}, (_, i) => `user${i}`);
        let clients = users.flatMap(user => [
            connect('http://localhost')
        ]);

        clients.forEach((client, index) => {
            client.emit('register', {username: users[index], password: 'password
                '}, (response) => {
                try {
                    assert.isTrue(response.success, `Registration failed for
                        user: ${users[index]}`);
                } catch (error) {
                    console.error('Command failed:', error.message);
                    done(error);
                    return;
                }
                if (index === users.length - 1) {
                    done();
                }
            });
```

```
        });
    });
});
```

Scalability Testing

- `import { connect } from 'socket.io-client';`
  - Imports the `connect` function from the `socket.io-client` library, which is used to establish socket connections to the server.
- `import { assert } from 'chai';`
  - Imports the `assert` function from the `chai` library, which is used for making assertions in tests.
- `this.timeout(30000);`
  - Increases the default timeout for the test suite to 30 seconds to accommodate potentially longer test execution times.
- `it('should handle 450 users', function(done) { ... });`
  - Defines a test case that checks if the chat application can handle 450 simultaneous users.
- `let users = Array.from({length: 450}, (_, i) => 'user${i}');`
  - Creates an array of 450 usernames, named `user0`, `user1`, ..., `user449`.
- `let clients = users.flatMap(user => [connect('http://localhost')]);`
  - Establishes socket connections to the server for each user and stores the connections in the `clients` array.
- `clients.forEach((client, index) => { ... });`
  - Iterates over each client connection to perform actions such as registration.
- `client.emit('register', {username: users[index], password: 'password'}, (response) => { ... });`
  - Emits a 'register' event for each client, sending the username and password to the server, and handles the server's response.
- `assert.isTrue(response.success, 'Registration failed for user: ${users[index]}');`

- Asserts that the registration response from the server is successful for each user. If not, logs an error.

- `if (index === users.length - 1) { done(); }`

  - Calls the `done` callback to signal the completion of the test once all users have been processed.

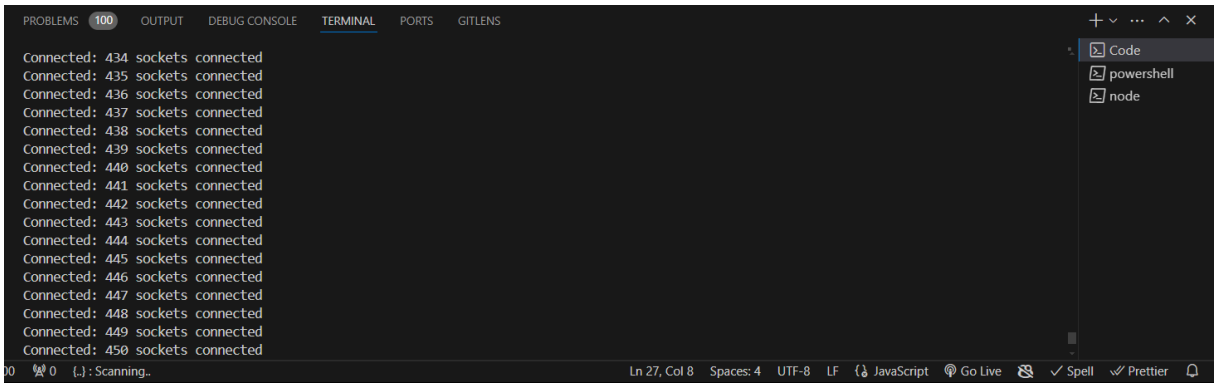

Figure 5: Connected users in the server terminal

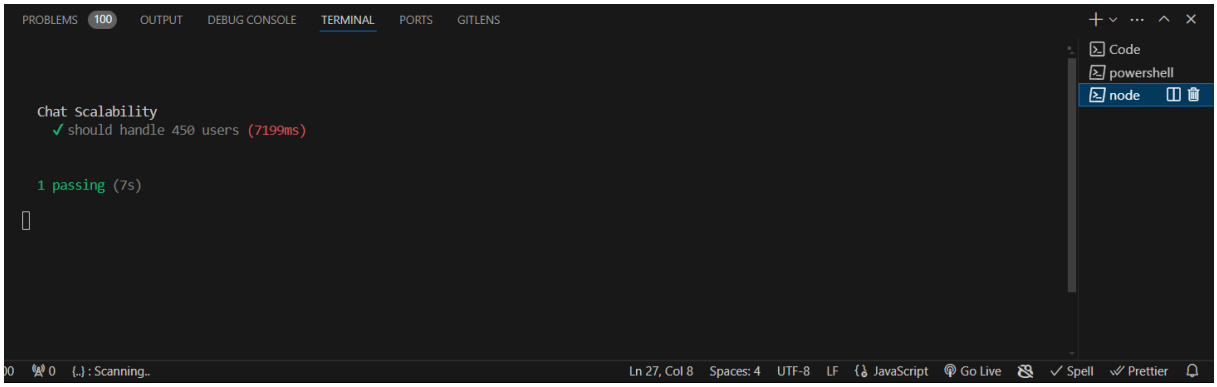

Figure 6: Result of the test

3.3 Historical Chat Access

Upon joining, new users will be able to retrieve past messages in the chat, allowing them to catch up on previous discussions.

```
1    // Connect Socket
2    connections.push(socket);
3    console.log('Connected: %s sockets connected', connections.length);
4    socket.emit('load all messages', chatStore);
```

Process of connect to socket

- **Add Socket to Connections**

  - `connections.push(socket);` adds the newly connected socket to the `connections` array to keep track of all active connections.

- **Log Number of Connections**

  - `console.log('Connected: %s sockets connected', connections.length);` logs a message to the console indicating the number of currently connected sockets using the length of the `connections` array.

- **Send Chat History to Client**

  - `socket.emit('load all messages', chatStore);` sends an event named 'load all messages' to the newly connected client, along with the `chatStore` array, which contains the chat history. The client can handle this event to load the chat history.

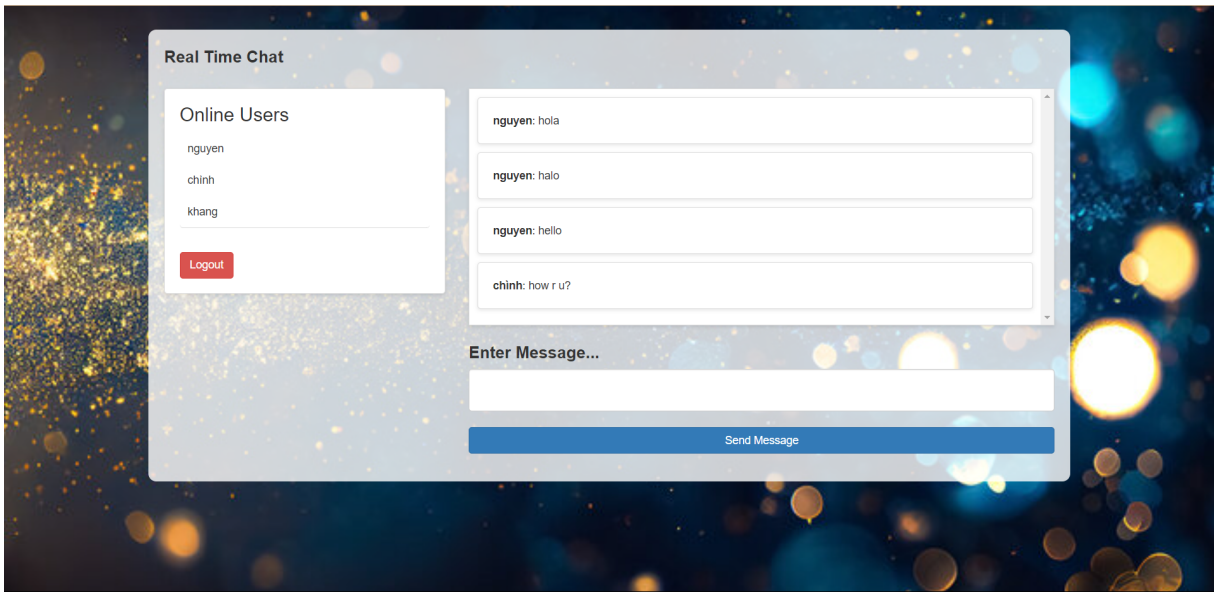

Figure 7: Historical chat access

- New user (Khang) is able to see the previous chat.

4 System Architectures

The system is using a centralized architecture. The system comprised of one main server and a backup server. Clients will connect to the main server to chat with each other in case of a main server failure, traffic will automatically be redirected to the backup server. Scalability can also be implemented by creating more server to balance the load to multiple server ensuring the application can handle a lot of user simultaneously.

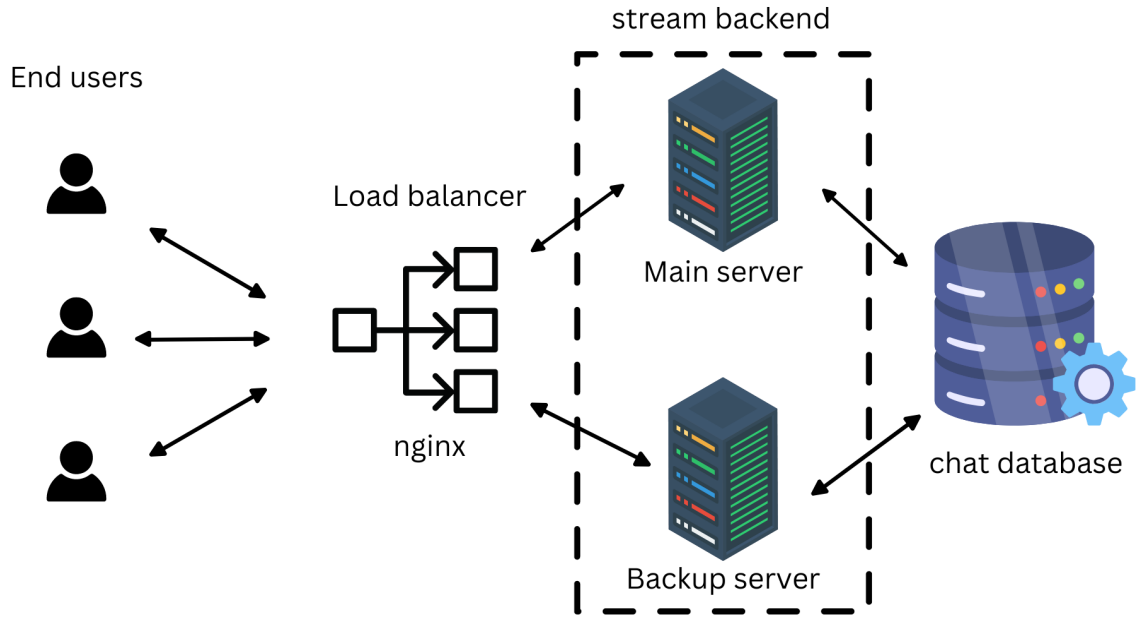

Figure 8: General System Architecture

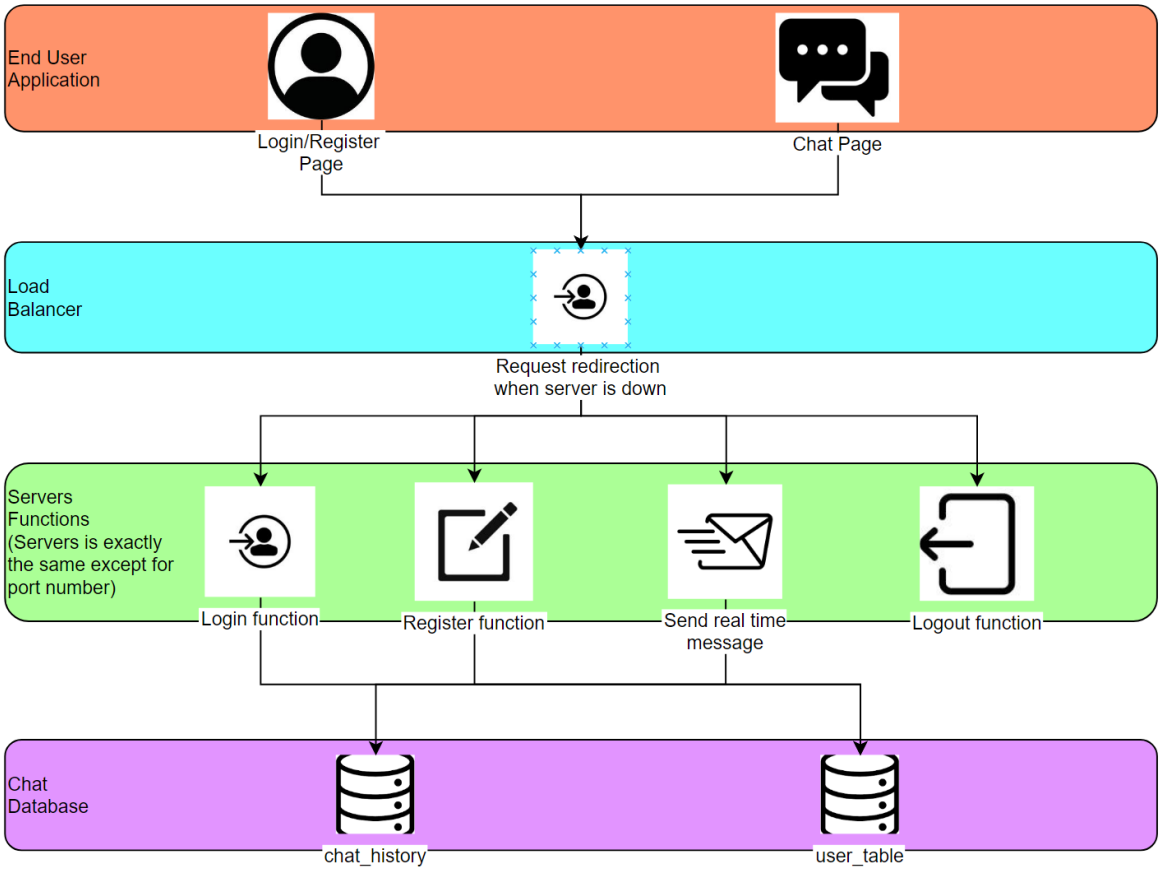

Figure 9: Layered System Architecture

5 Conclusion

The real-time chat application described in this report aims to provide a robust and user-friendly platform for instant communication. By meeting both functional and non-functional requirements, the application ensures a reliable and enjoyable user experience. The system's architecture supports scalability, security, and some fault tolerances, making it a dependable solution for real-time messaging needs.