Team: E - Bingo

Team members:

Ekarat Buddharuksa
Danish Nguyen
Mya Phyu
Karl Xavier Arcilla

Team's repo: https://github.com/sfsu-csc-667-spring-2024-roberts/term-project-team-e-bingo

Team Collaboration and Schedule

Group Meetings:
Our team meets regularly, typically 1-2 times per week, with each meeting lasting about one to two hours. These meetings are crucial for discussing our progress, planning, and distributing tasks among team members.

Task Distribution and Execution:
In our meetings, we create a to-do list that outlines the essential tasks for our project. Team members then choose tasks to complete on their own, usually finishing them within 1-2 days. This method helps us keep the project moving forward steadily.

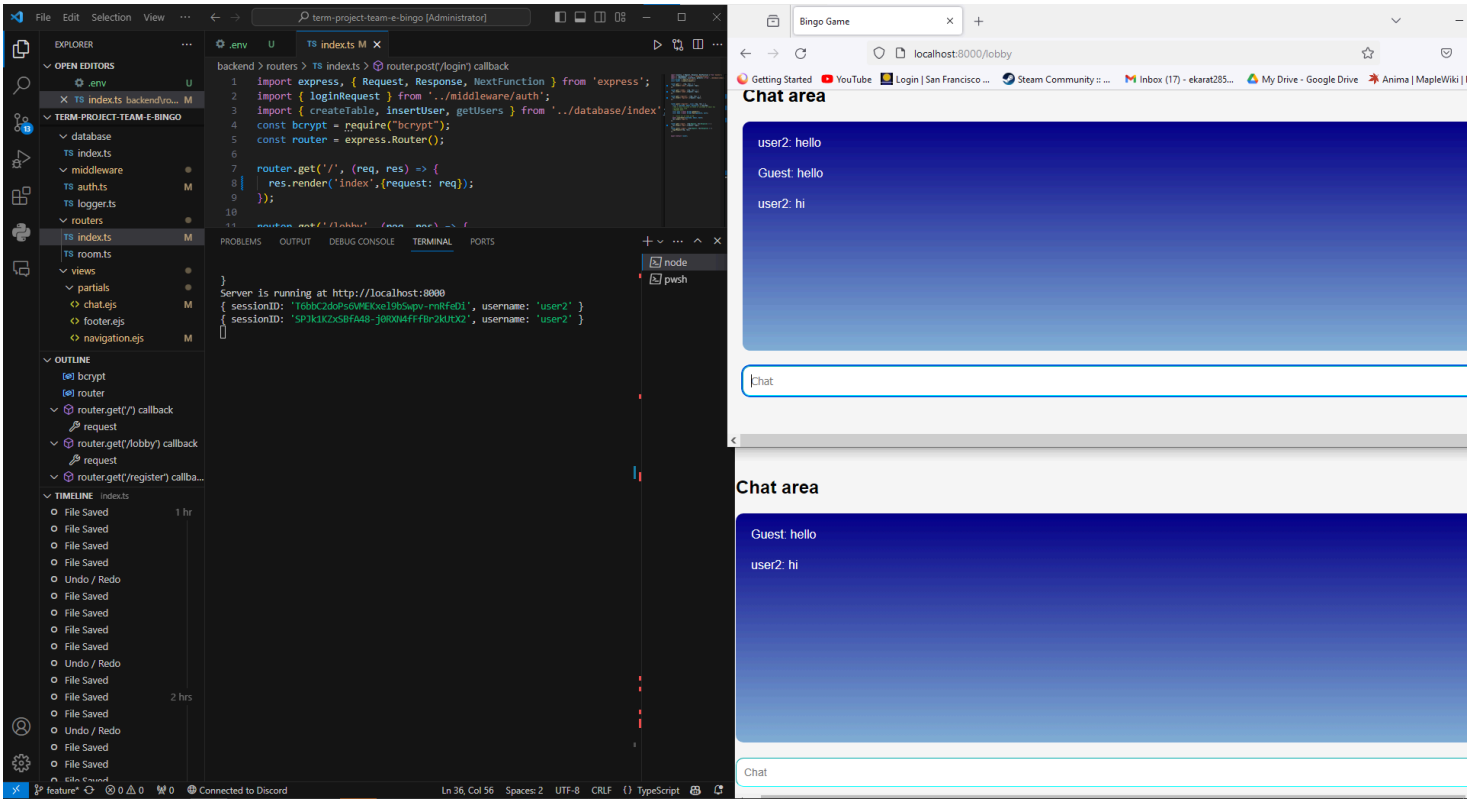
Current Goals and Milestones:
We are currently focused on developing game functionality and adding chat features in both the lobby and game room. We plan to work on these key components in our upcoming sessions, ensuring they are well-integrated and functional.

Game description

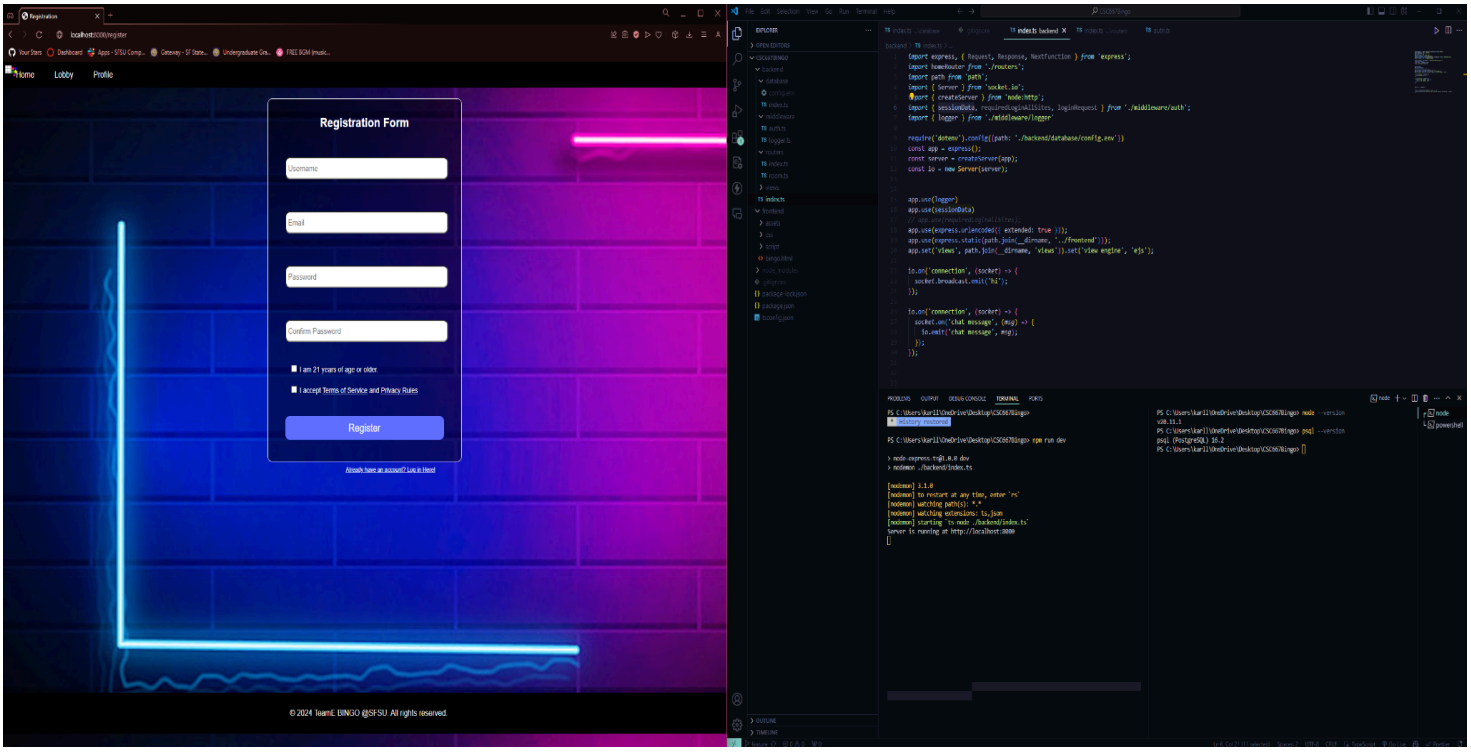
Bingo is a classic, easy-to-play game that can be enjoyed by people of all ages. In our application, the maximum number of players allowed at one time is four. Each player receives a bingo card, which is a 5x5 grid filled with random numbers. Numbers are drawn randomly from a pool ranging from 1 to 75 and announced to the players. These numbers will also be displayed on the side of the screen for rechecking. Players mark these numbers on their cards if they appear. The goal of the game is to be the first to complete a pattern on the card, which includes a straight line horizontally, vertically, or diagonally. When a player achieves the required pattern, they can press the "Bingo!" button to stop the game. Their card is then checked for accuracy, and if correct, they are declared the winner.

Team member's test page

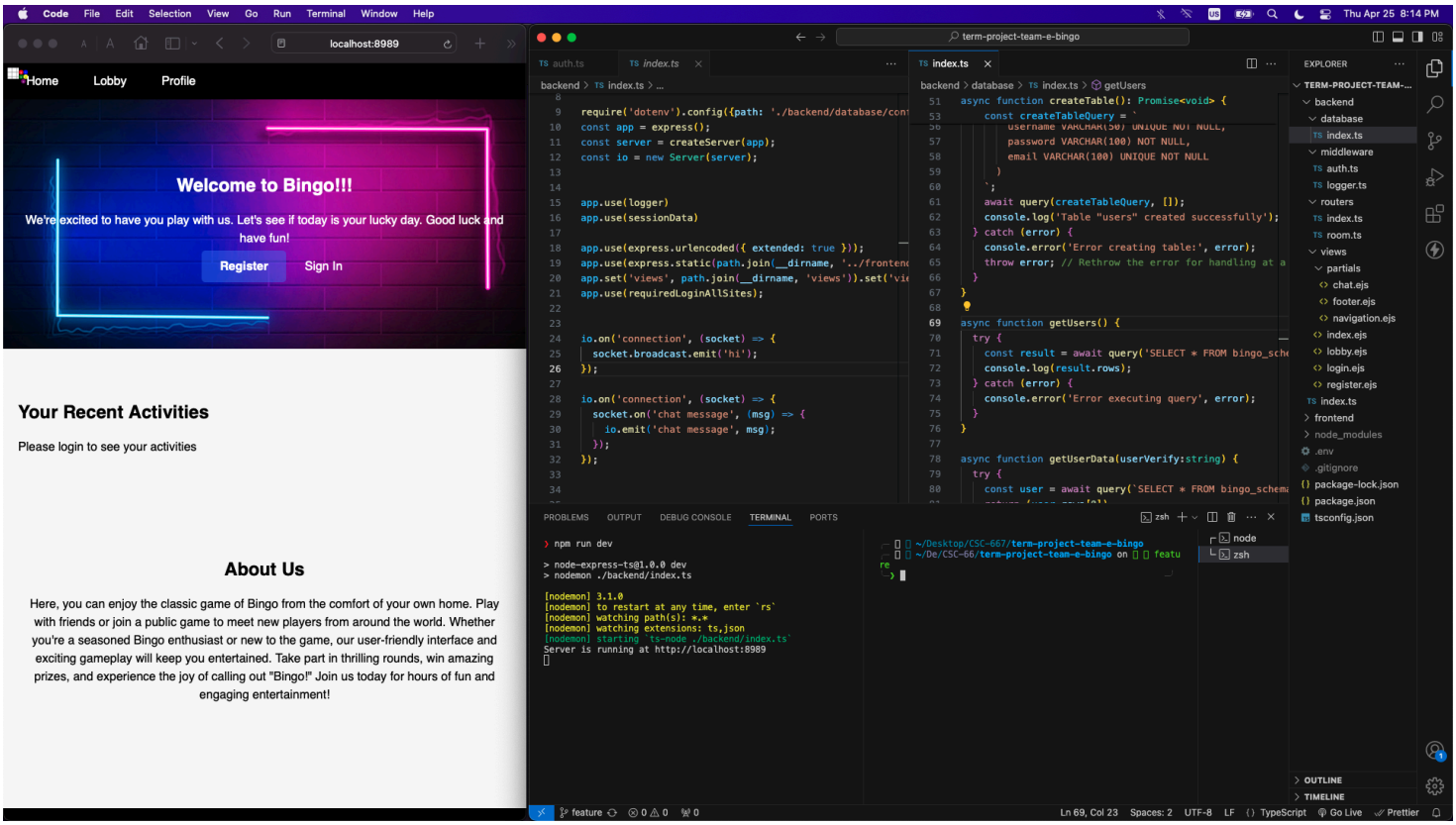
Ekarat Buddharuksa



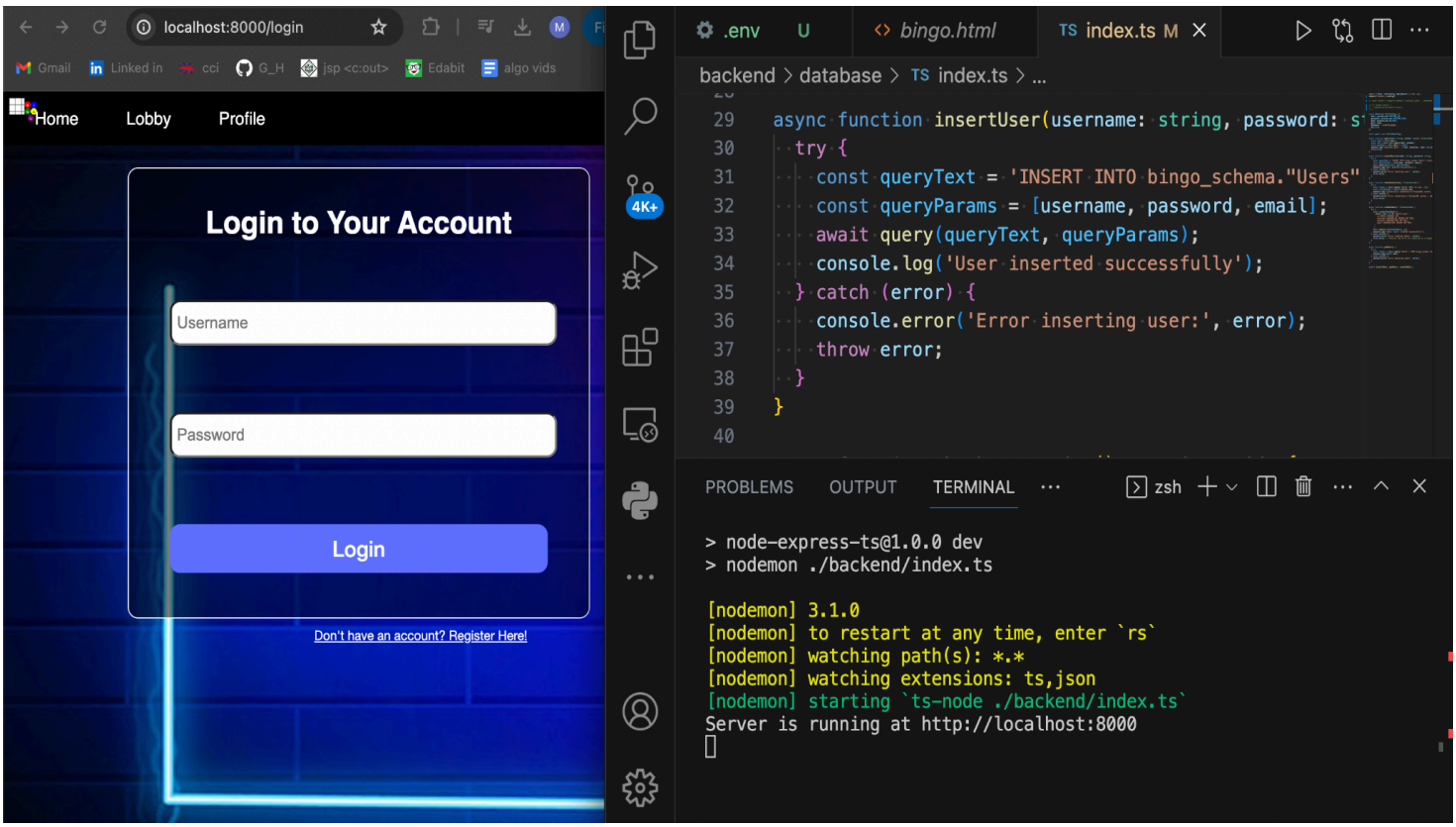
Karl Xavier Arcilla



Danish Nguyen



Mya Phyu



Authentication

Authentication on accessing the page

```
const requiredLoginAllSites = (req: Request, res: Response, next: NextFunction) => {
    !['/login', '/register', '/' ].includes(req.path) ? isAuthenticated(req, res, next) : next();
}
```

```
const isAuthenticated = (req: Request, res: Response, next: NextFunction) => {
    if (req.session.user) {
        next();
    } else {
        res.redirect('/login');
    }
}
```

```
app.use(requiredLoginAllSites);
```

In the middleware setup, we first check if the current path is one of the unrestricted routes: ‘/login’, ‘/register’, or the 'home page ‘/’. These routes are accessible to all users without the need for authentication. If the request does not direct to one of these routes, the middleware then checks for user authentication by examining req.session.user. This session variable holds the user's login information if they are authenticated. If req.session.user exists, it indicates that the user is logged in, and the session contains their information. In cases where req.session.user is not found, it suggests that the user is not logged in or that their session has expired. Consequently, the application redirects the user to the login page to ensure that only authenticated users can access sensitive or restricted areas of the application.

Authentication on login

```
router.post('/login', (req:Request, res:Response) => {
  loginRequest(req, res);
});
```

```
const loginRequest = async (req: Request, res: Response) => {
    const sessionID = req.session.id;

    const {username, password} = req.body
    if (!username || !password) {
        return res.render('login', {errorMessage: 'Please log in to access the game!'})
    }

    if (await authenticate(username, password)) {
        req.session.user = { sessionID , username }
        return res.redirect('/')
    } else {
        return res.render('login', {errorMessage: 'Invalid username or password'})
    }
}
```

```
const authenticate = async (username: string, password: string): Promise<boolean> => {
    try {
      const dbUser = await getUserData(username);
      return await bcrypt.compare(password, dbUser.password);
    } catch (err) {
      console.error(err);
      return false;
    }
}
```

```
async function getUserData(userVerify:string) {
  try {
    const user = await query(`SELECT * FROM bingo_schema."Users" WHERE username = $1`, [userVerify]);
    return (user.rows[0])
  } catch (error) {
    console.error('Error executing query', error);
  }
}
```

The login process initiates with a POST request to the '/login' endpoint, activating the loginRequest function. This function checks for the presence of username and password in the request. If either is missing, it immediately renders the login page with a prompt for complete credentials. Provided both fields are present, loginRequest invokes the authenticate function, which uses getUserData to fetch the user's data from a PostgreSQL database. If the user exists, bcrypt compares the provided password against the stored hash. A successful match updates the session with the user's ID and username, marking them as authenticated and redirecting them to the home page. Failure due to incorrect credentials or database errors results in the login page being re-rendered with an error message.

How password is stored in database

```
router.post('/register', async (req, res) => {
  const { username, email, password } = req.body;
  const salt = await bcrypt.genSalt();
  const hash = await bcrypt.hash(password, salt);

  await insertUser(username, hash, email);
  res.render('login',{session: req.session});
});
```

```
async function insertUser(username: string, password: string, email: string): Promise<void> {
  try {
    const queryText = 'INSERT INTO bingo_schema."Users" (username, password, email) VALUES ($1, $2, $3)';
    const queryParams = [username, password, email];
    await query(queryText, queryParams);
    console.log('User inserted successfully');
  } catch (error) {
    console.error('Error inserting user:', error);
    throw error;
  }
}
```

Upon user registration, the provided password is secured using bcrypt, which generates a salt and hashes the password to safeguard user information. The insertUser function then constructs a parameterized SQL query to insert the username, hashed password, and email into the bingo_schema."Users" table within the PostgreSQL database. Parameterized queries, which use placeholders like $1, $2, and $3, protect against SQL injection by separating the data from the SQL logic. Once the user details are successfully inserted into the database, the user is redirected to the login page with their sensitive information securely stored.

Below are examples of user data stored in the database

```
executed query { text: 'SELECT * FROM bingo_schema."Users"', duration: 417, rows: 4 }
[
  {
    user_id: 7,
    username: 'testAccount3',
    password: '$2b$10$bN4eK5w9i7FEkYv3DI5vNO4.nZcpi2J2kis/TtnrSWDYWBfax/Fxe',
    email: 'testAccount3@test.com'
  },
  {
    user_id: 8,
    username: 'hello',
    password: '$2b$10$7fnK.093bFniPQ4rk6NGpOXHUA56XIUZH4NmYmhrH6DiSynGowMSW',
    email: 'hello@sfsu'
  },
```

Session management.

```
const sessionData = session({
    secret: process.env.SECRET_KEY_BINGO || 'your-secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: {
        secure: false,
        httpOnly: true,
        maxAge: 1 * 10 * 60 * 1000,
    },
})
```

```
declare module 'express-session' {
    interface SessionData {
        user?: { sessionID:string, username:string,
        }
    }
}
```

The session in our application is managed by express-session middleware. When initializing sessionData, a secret key is set to sign session ID cookies, with resave and saveUninitialized options set to false to prevent saving unmodified or new but not modified sessions. Cookies are configured to be non-secure (for development purposes) and HTTP-only to prevent client-side script access. They expire after 10 minutes (for development purposes). When the session expires, the user is automatically logged out and needs to be logged in to use the site further.

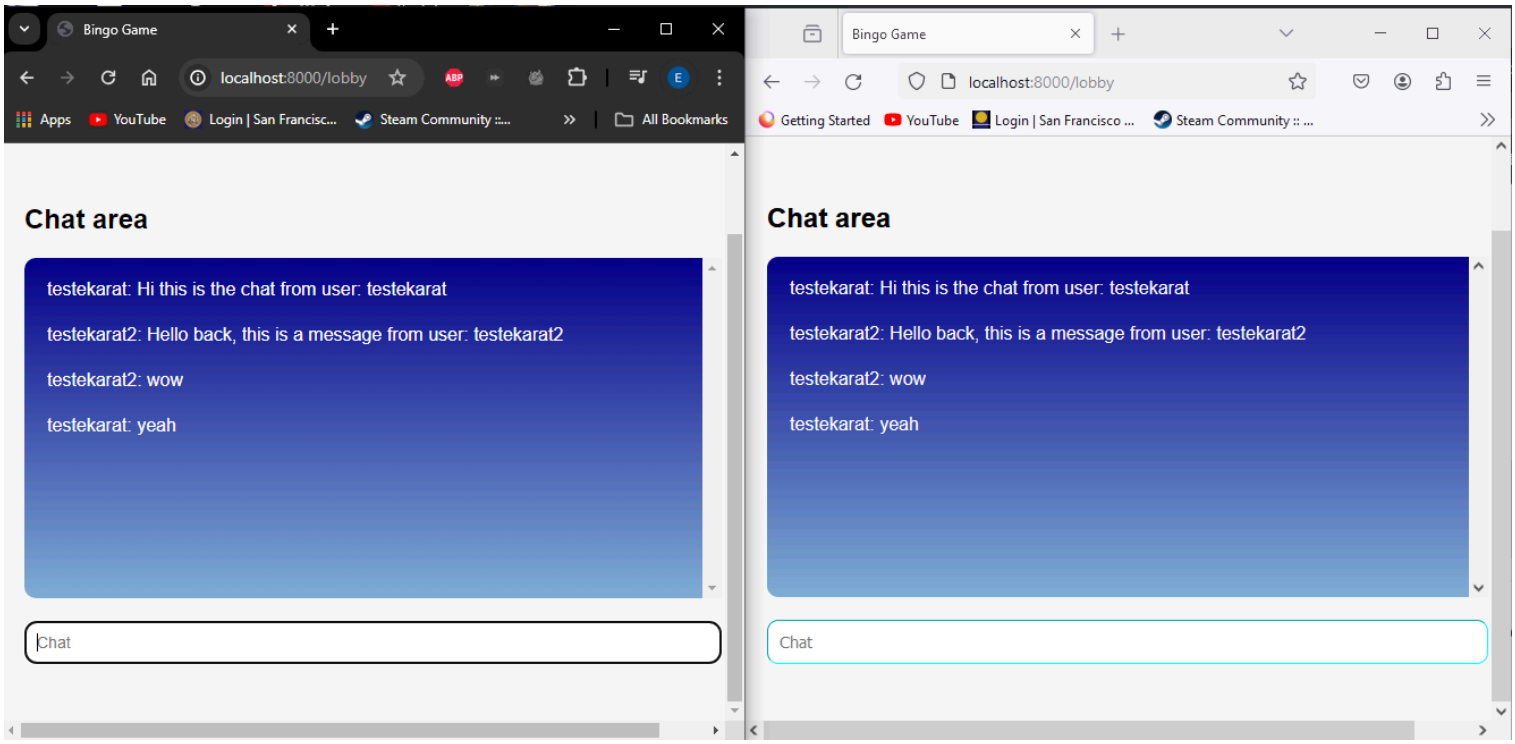
During login, if the authentication is successful, the session is populated with the user's session ID and username.. This session object, including the user information, is what the application uses to maintain user state across requests.

Upon logout, the session got destroyed along with the cookie store in the client browser.

```
router.post('/logout', (req:Request, res:Response) => {
  req.session.destroy((err) => {
    if (err) {
      return res.redirect('/');
    }
    res.clearCookie('connect.sid');
    res.redirect('/login');
  });
});
```

Chat

Example of two user use chat functionality



In the example provided, the chat functionality within the application is accomplished by Socket.IO, a real-time bidirectional event-based communication library. When a user sends a chat message, the data—including the username and message content—is transmitted to the server via a Socket.IO event. The server-side Socket.IO listener receives this event and processes the incoming data. It then emits a new event to broadcast the chat message to all connected clients, ensuring that every user in the application receives the update in real-time. This allows for a dynamic chat experience where messages are shared immediately among all users, facilitating interactive conversations within the application.