

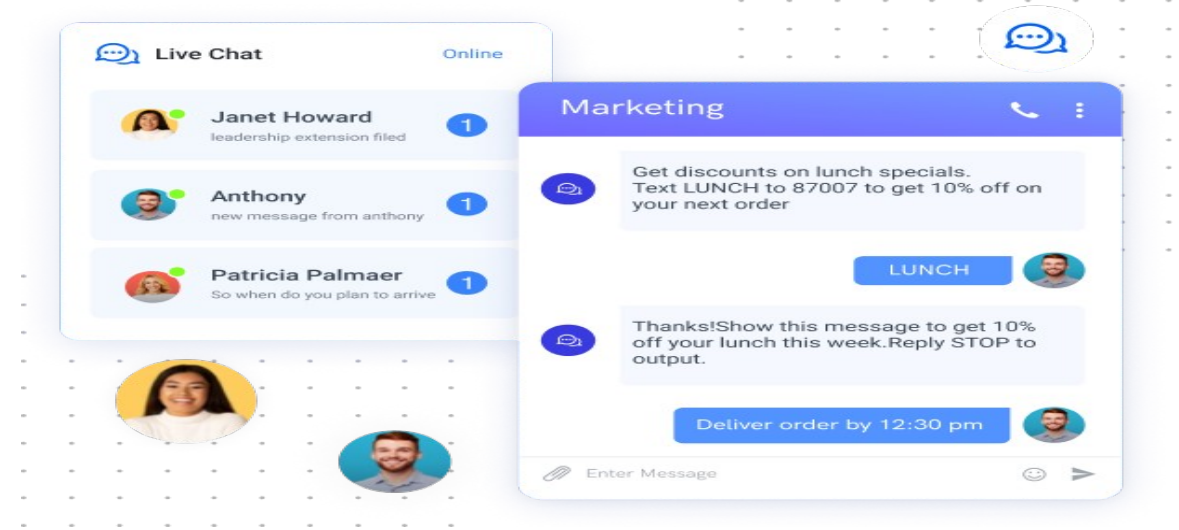
Advanced Real-Time Chat Application using CORE JAVA

NAME- RN Umashankar
COLLEGE- REVA University
SEMESTER 7th
B TECH (ECE)

INTRODUCTION

- The development of this application is not just a technical endeavor; it represents an opportunity to create a meaningful tool that can enhance communication. In a world where remote work and virtual interactions are becoming the norm,
- effective communication tools are essential for fostering collaboration and building relationships.
- By focusing on user experience, security, and functionality, this project aims to set a new standard for chat applications,
- demonstrating how technology can facilitate better connections among users. This application will also serve as a learning experience in full-stack development, providing insights into real-time data handling, user authentication, and responsive design.

OBJECTIVES



1. User Authentication:

- 1. Secure Registration:** Implement a robust registration system that requires users to provide a unique username and a strong password,
2. ensuring that user credentials are securely stored using hashing algorithms.
- 3. Login Functionality:** Enable users to log in using their credentials,
4. validating them against stored data and issuing a JSON Web Token (JWT) for session management.
- 5. Session Management:** Maintain user sessions securely, ensuring users remain logged in until they choose to log out.

- Public Chat Rooms:**

- Create Public Rooms:** Allow users to create and join public chat rooms based on various topics of interest, fostering community discussions.

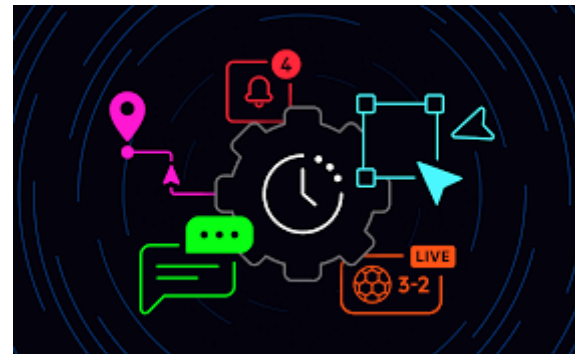
- Room Management:** Implement features to display available chat rooms, allowing users to navigate and join them easily.

- Message Broadcasting:** Facilitate real-time message broadcasting in public chat rooms, ensuring all participants receive messages instantly.

- **Private Messaging:**
- **Direct Messaging:** Provide functionality for users to send direct messages to other users,
- enabling one-on-one conversations.
- **User List:** Implement a user list feature that displays online users,
- making it easy to initiate private chats.
- **Message Notifications:** Ensure users receive notifications for new private messages,
- enhancing engagement.

- **Message History:**
- **Persistent Storage:** Store chat messages in a MongoDB database,
 - allowing users to access their chat history across sessions.
- **Retrieve Messages:** Implement functionality to retrieve and display previous messages
 - when users enter a chat room or reopen a private conversation.
- **Search Functionality:** Allow users to search their chat history for specific keywords or messages, improving usability.

- User Profiles:**
- Profile Customization:** Allow users to create and customize their profiles with information such as display name, profile picture, and status messages.
- User Presence Indicators:** Implement indicators to show when users are online or offline, facilitating better communication.
- Real-Time Functionality:**
- Socket.IO Integration:** Utilize Socket.IO for real-time communication, enabling instant message delivery and updates without page refreshes.
- Event Handling:** Implement event listeners for various actions (sending messages, joining rooms) to ensure smooth interactions.



METHODOLOGY

- **Technology Stack Selection:**
- **Frontend:** Choose technologies such as HTML, CSS, and JavaScript frameworks (e.g., React or Vue.js) for building the user interface.
- **Backend:** Select Node.js as the server-side runtime environment, with Express.js for handling HTTP requests and Socket.IO for real-time communication.
- **Database:** Opt for MongoDB as the NoSQL database for storing user data and chat messages due to its flexibility and scalability.
- **Authentication:** Use JWT (JSON Web Tokens) for secure user authentication and session management.

•**System Design:**

- Architecture Design:** Outline the overall architecture, including client-server interaction and the flow of data between components.

- Database Schema Design:**

- User Schema:** Define fields such as username, password (hashed), profile picture URL, and status.

- Message Schema:** Define fields for sender ID, recipient ID (for private messages), message content, timestamp, and chat room ID (for public messages).

- Wireframing:** Create wireframes or mockups of the user interface to visualize the layout and user experience.

- **Backend Development:**

- **Set Up Server:**

- Initialize a Node.js project and install required packages (Express.js, Socket.IO, Mongoose for MongoDB).
- Create the server structure, including routes for authentication and message handling.

- **Implement User Authentication:**

- Create routes for user registration and login, implementing password hashing with bcrypt.js.
- Set up JWT for session management, ensuring secure token issuance upon successful login.

- **Frontend Development:**
- **Build User Interface:**
 - Create the layout using HTML and CSS, ensuring a responsive design that adapts to various screen sizes.
 - Use JavaScript (or a framework) to handle user interactions, such as sending messages and navigating chat rooms.
- **Integrate Socket.IO:**
 - Set up the Socket.IO client to handle real-time messaging, connecting to the server and listening for incoming messages.
 - Implement functionality to emit messages from the client and display them in the chat interface.

- Unit Testing:** Write unit tests for individual functions and components, ensuring that they work as expected.
- Integration Testing:** Test the interaction between different parts of the application (e.g., authentication flow, message sending).
- User Acceptance Testing:**
- Acceptance Testing:** Conduct testing with real users to gather feedback on usability and functionality. Identify any issues or areas for improvement.
- Deployment:**
- Choose Hosting Service:** Select a cloud service provider (e.g., Heroku, AWS, or DigitalOcean) to host the application.
- Set Up Environment:** Configure the production environment, including database connections and environment variables.
- Continuous Integration/Continuous Deployment (CI/CD):** Implement CI/CD practices to automate testing and deployment processes.
- Monitoring and Maintenance:**
- Performance Monitoring:** Use monitoring tools to track application performance, user activity, and error logs.
- Regular Updates:** Schedule regular updates to address bugs, enhance features, and improve security.
- User Support:** Provide documentation and support channels for users to report issues or seek assistance.

Working of the Advanced Real-Time Chat Application

- **User Interface (UI)**
- **Landing Page:**
 - Users arrive at the landing page where they can either register or log in to access the chat functionalities.
- **Authentication:**

Registration: New users fill out a registration form with a username and password.

- Upon submission, the application validates the input and stores the user data in the database (hashed password for security).
- **Login:** Existing users enter their credentials. The application verifies the credentials, and if valid, issues a JSON Web Token (JWT) to manage the session.

Real-Time Messaging

- Socket.IO Connection:**

- Upon logging in, the client establishes a connection to the server via Socket.IO, enabling real-time communication.

- The client listens for events from the server and emits events to send messages.

- Joining Chat Rooms:**

- Users can join public chat rooms. When they do, the application emits a joinRoom event to the server, notifying it of the user's presence in that room.

3. Server-Side Processing

- **Message Handling:**

- The server listens for incoming message events. Upon receiving a message:
 - The server stores the message in the MongoDB database for persistence.
 - It broadcasts the message to all users in the relevant chat room via Socket.IO, ensuring all users see the new message in real-time.

- **Private Messaging:**

- For private messages, the server routes the message directly to the intended recipient, ensuring only the sender and recipient can view the message.

4. Message History

- **Retrieving Messages:**

- When a user joins a chat room, the application queries the MongoDB database to retrieve the chat history for that room.
- The retrieved messages are displayed in the chat interface, allowing users to see past conversations.

User Presence and Notifications

User List:

The application maintains a list of online users in each chat room. When a user joins or leaves, the server updates this list and notifies all clients.

Notifications:

Users receive real-time notifications for new messages, enhancing engagement and ensuring they do not miss important communications.

6. Responsive Design

Cross-Device Compatibility:

The application's frontend is designed to be responsive, allowing users to access the chat from desktops, tablets, and mobile devices. This is achieved using flexible layouts and media queries in CSS.

- **Security Features**

- **Data Security:**

- All communications between the client and server are secured using HTTPS, protecting user data during transmission.
- User inputs are validated and sanitized to prevent security vulnerabilities like XSS and SQL injection.

- **8. Logging Out**

- **Session Termination:**

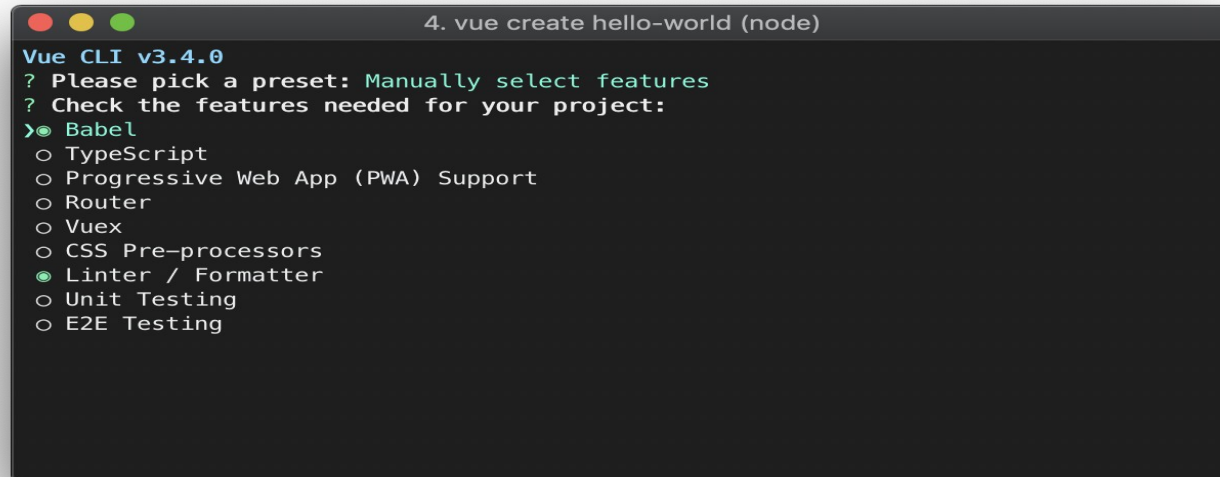
- When users choose to log out, the application clears the JWT from the client's storage,
- effectively terminating their session and redirecting them to the landing page.

CODE EXPLANATION (PROJECT STRUCTURE)

- /chat-app
 - /public
 - index.html
 - style.css
 - script.js
 - /server
 - server.js
 - models
 - User.js
 - Message.js
 - package.json

INITIALIZE THE PROJECT

- mkdir chat-app
- cd chat-app
- npm init -y
- npm install express socket.io mongoose bcryptjs jsonwebtoken cors dotenv

A terminal window with a dark background and light text. The title bar at the top reads "4. vue create hello-world (node)". The terminal content shows the Vue CLI v3.4.0 prompt, followed by instructions to pick a preset and check features. A list of features is displayed with radio buttons, where "Babel" and "Linter / Formatter" are selected.

```
Vue CLI v3.4.0
? Please pick a preset: Manually select features
? Check the features needed for your project:
> Babel
  TypeScript
  Progressive Web App (PWA) Support
  Router
  Vuex
  CSS Pre-processors
  Linter / Formatter
  Unit Testing
  E2E Testing
```

Create the User Model (server/models/User.js)

- `// server/models/User.js`
- `const mongoose = require('mongoose');`
- `const userSchema = new mongoose.Schema`
- `{`
- `username: { type: String, required: true, unique: true },`
- `password: { type: String, required: true },`
- `});`
- `module.exports = mongoose.model('User', userSchema);`

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World');
})

var server = app.listen(8081, function () {
  var host = server.address().address
  var port = server.address().port

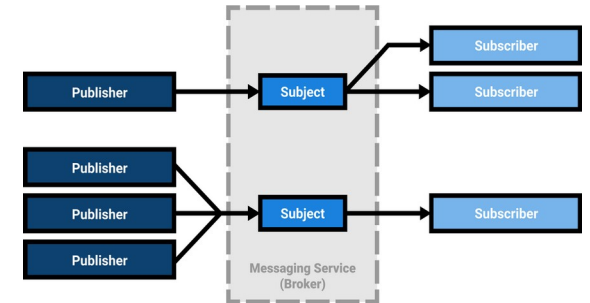
  console.log("Example app listening at http://%s:%s", host, port)
})
```

CREATING THE MESSAGE MODLE

```
// server/models/Message.js
const mongoose = require('mongoose');

const messageSchema = new mongoose.Schema
({
  sender:
  { type: mongoose.Schema.Types.ObjectId, ref: 'User', required:
  true },
  recipient:
  { type: mongoose.Schema.Types.ObjectId, ref: 'User', required:
  true },
  content:
  { type: String, required: true },
  timestamp:
  { type: Date, default: Date.now },
});

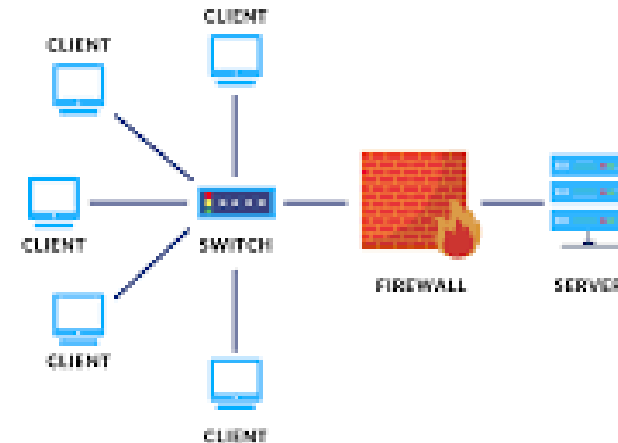
module.exports = mongoose.model('Message', messageSchema);
```



SETTING UP THE SERVER

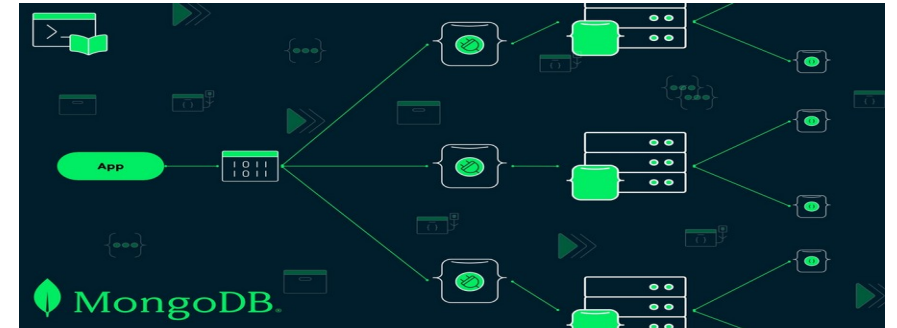
```
•// server/server.js
•const express = require('express');
•const http = require('http');
•const socketio = require('socket.io');
•const mongoose = require('mongoose');
•const bcrypt = require('bcryptjs');
•const jwt = require('jsonwebtoken');
•const cors = require('cors');
•require('dotenv').config();

•const User = require('./models/User');
•const Message = require('./models/Message');
```



```
const app = express();
const server = http.createServer(app);
const io = socketIo(server);
```

```
app.use(cors());
app.use(express.json());
app.use(express.static('public'));
```



CONNECTING TO MONGO DB

```
mongoose.connect(process.env.MONGODB_URI, { useNewUrlParser: true, useUnifiedTopology: true }); //
```

```
Register User app.post('/register', async (req, res) =>
```

```
  { const { username, password } = req.body;
```

```
  const hashedPassword = await bcrypt.hash(password, 10);
```

```
  const newUser = new User({ username, password: hashedPassword });
```


```
  await newUser.save();
```

```
  res.status(201).send('User registered');
```

```
});
```

LOGIN USER

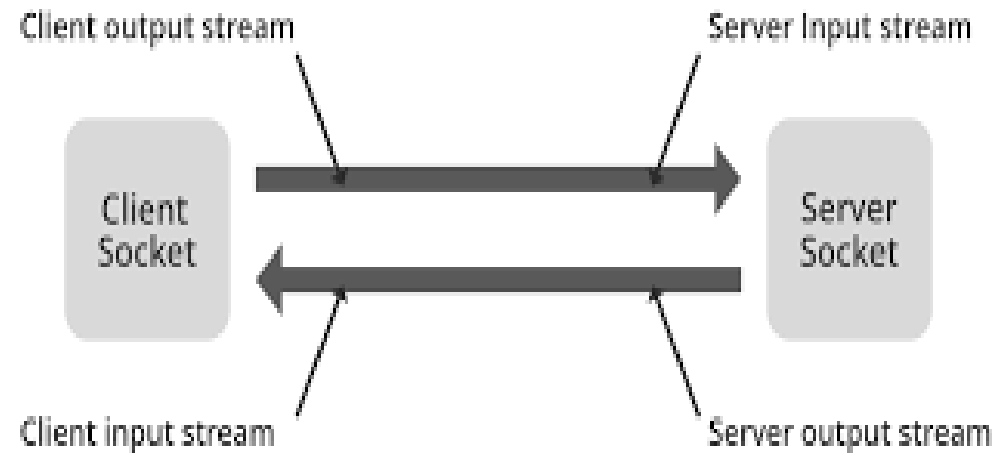
- `// Login User app.post('/login', async (req, res) =>`
- `{ const { username, password } = req.body;`
- `const user = await User.findOne({ username });`
- `if (user && (await bcrypt.compare(password, user.password)))`
- `{ const token = jwt.sign({ id: user._id },`
- `process.env.JWT_SECRET); res.json({ token });`
- `}`
- `else`
- `{ res.status(401).send('Invalid credentials');`
- `}`
- `}`
- `);`



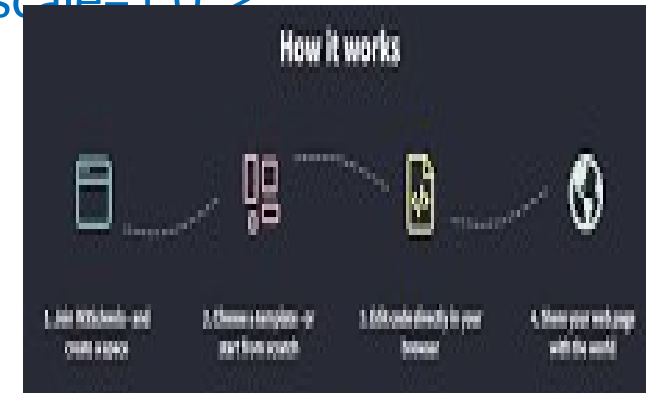
A screenshot of a web application window titled "User Login". The window has a dark blue background. At the top, the text "User Login" is displayed in white. Below this, the text "My Account" is written in a large, bold, yellow font. Underneath, there are two input fields. The first is labeled "Enter Your Username" and has a yellow icon of a person to its left. The second is labeled "Enter Your Password" and has a yellow icon of a padlock to its left. Below the password field, there is a yellow button with the text "Login". At the bottom of the form, there is a link that says "Don't have an account ?" followed by a yellow button with the text "Sign Up".

HANDLING THE SOCKET CONNECTION

- `// Handle socket connections io.on('connection', (socket) =>`
- `{ console.log('New user connected');`
- `socket.on('join', (userId) => {`
- `socket.join(userId);`
- `}); socket.on('chatMessage', async (msg) =>`
- `{`
- `const message = new Message(msg);`
- `await message.save();`
- `io.to(msg.recipient).emit('chatMessage', msg); });`
- `socket.on('disconnect', () =>`
- `{`
- `console.log('User disconnected');`
- `});`
- `});`
- `// Start the server const PORT = process.env.PORT || 3000; server.listen(PORT, () => { console.log(`Server is running on http://localhost:${PORT}`); });`



- <html lang="en">
- <head>
 - <meta charset="UTF-8">
 - <meta name="viewport" content="width=device-width, initial-scale=1.0">
 - <title>Chat App</title>
 - <link rel="stylesheet" href="style.css">
- </head>
- <body>
 - <div class="container">
 - <h1>Chat Application</h1>
 - <div id="auth">
 - <form id="registerForm">
 - <input type="text" id="registerUsername" placeholder="Username" required>
 - <input type="password" id="registerPassword" placeholder="Password" required>
 - <button type="submit">Register</button>



CREATING FRONT END USING HTML

```
• </form>
•   <form id="loginForm">
•     <input type="text" id="loginUsername" placeholder="Username" required>
•     <input type="password" id="loginPassword" placeholder="Password" required>
•     <button type="submit">Login</button>
•   </form>
• </div>
• <div id="chat" style="display: none;">
•   <ul id="messages"></ul>
•   <form id="form">
•     <input id="input" autocomplete="off" placeholder="Type a message..." />
•     <button>Send</button>
•   </form>
• </div>
• </div>
• <script src="/socket.io/socket.io.js"></script>
• <script src="script.js"></script>
• </body>
• </html>
```

ADDING STYLE USING CSS

- `/* public/style.css */`
- `body {`
- `font-family: Arial, sans-serif;`
- `background-color: #f4f4f4;`
- `} .container`
- `{`
- `width: 400px;`
- `margin: 50px auto; padding: 20px;`
- `background: #fff;`
- `border-radius: 5px;`
- `box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);`
- `}`
- `#chat { display: none; }`
- `#messages { list-style-type: none; padding: 0; height: 200px; overflow-y: scroll; border: 1px solid #ccc; }`

Code

CSS

<>

≡

```
/* image 6 */
```

```
position: absolute;
```

```
width: 873px;
```

```
height: 796px;
```

```
left: 1817px;
```

```
top: 134px;
```

```
background: url(image.png);
```

CONTUATION

- #messages li
- {
- padding: 5px 10px;
- border-bottom: 1px solid #eee;
- }
- form
- {
- display: flex;
- }
- form input
- {
- flex: 1; padding: 10px;
- }
- form button
- {
- padding: 10px;
- }

Edit Style Definition

Provide a name and a list of CSS classes for this style.

NOTE: Modifying a style definition does not change existing places in your site where the style has already been used. It affects only the CSS classes applied when the style is chosen in the future.

Add Style Definition

Provide a name and a list of CSS classes for this style.

NOTE: Modifying a style definition does not change existing places in your site where the style has already been used. It affects only the CSS classes applied when the style is chosen in the future.

Style Name:
Enter a descriptive name for the style (e.g. Large Orange Button).

CSS Classes:
Enter one or more space-separated CSS class names to be applied when this style is chosen. Note that CSS class names **are** case sensitive.

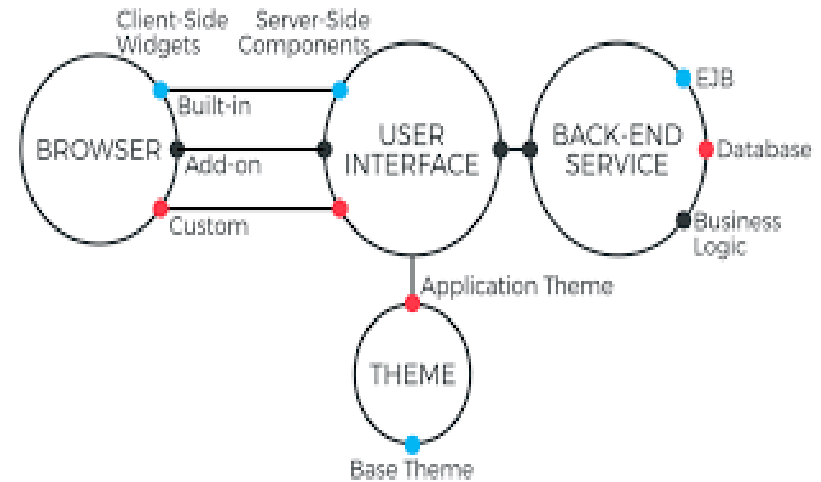
Category:

Description:
Enter a description for the style (optional).

[Save & Add New](#) [Save](#) [Cancel](#)

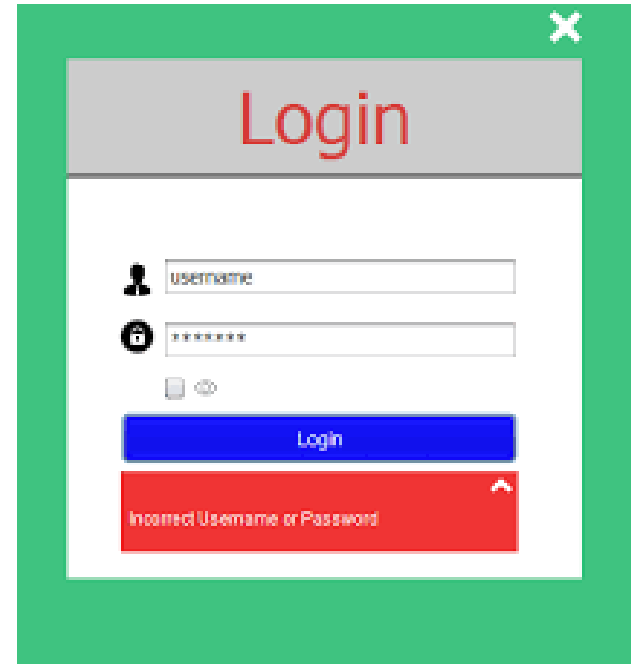
Add Client-Side Logic (public/script.js)

- `const socket = io();`
- `// Registration`
- `document.getElementById('registerForm').onsubmit = (e) =>`
- `{`
- `e.preventDefault();`
- `const username = document.getElementById('registerUsername').value;`
- `const password = document.getElementById('registerPassword').value;`
- `fetch('/register',`
- `{`
- `method: 'POST',`
- `headers: { 'Content-Type': 'application/json' },`
- `body: JSON.stringify({ username, password }),`
- `})`
- `.then(response => alert(response.ok ? 'Registered!' : 'Failed to register'));`
- `};`



```
// Login
document.getElementById('loginForm').onsubmit = (e) =>
{
  e.preventDefault(); const username = document.getElementById('loginUsername').value;
  const password = document.getElementById('loginPassword').value;

  fetch('/login',
  {
    method: 'POST',
    headers: {'Content-Type': 'application/json' },
    body: JSON.stringify({ username, password }),
  })
  .then(response =>
  {
    if (response.ok)
    {
      document.getElementById('auth').style.display = 'none';
      document.getElementById('chat').style.display = 'block';
      socket.emit('join', username);
    } else
    {
      alert('Login failed');
    }
  });
};
```



Login

username

password

Login

Incorrect Username or Password

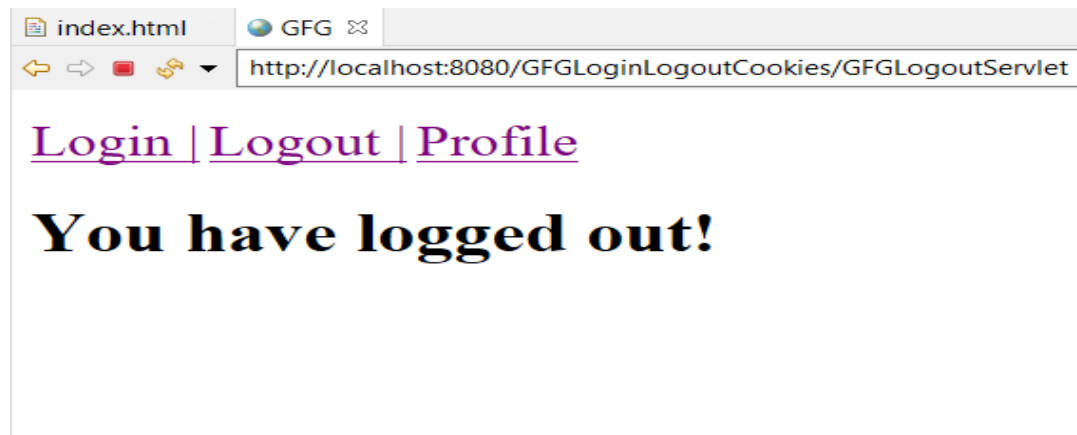
SENDING AND RECEIVING MESSAGE

- `// Send message`
- `document.getElementById('form').onsubmit = (e) => {`
- `e.preventDefault();`
- `const message = document.getElementById('input').value; socket.emit('chatMessage', { content: message });`
- `document.getElementById('input').value = '';`
- `};`

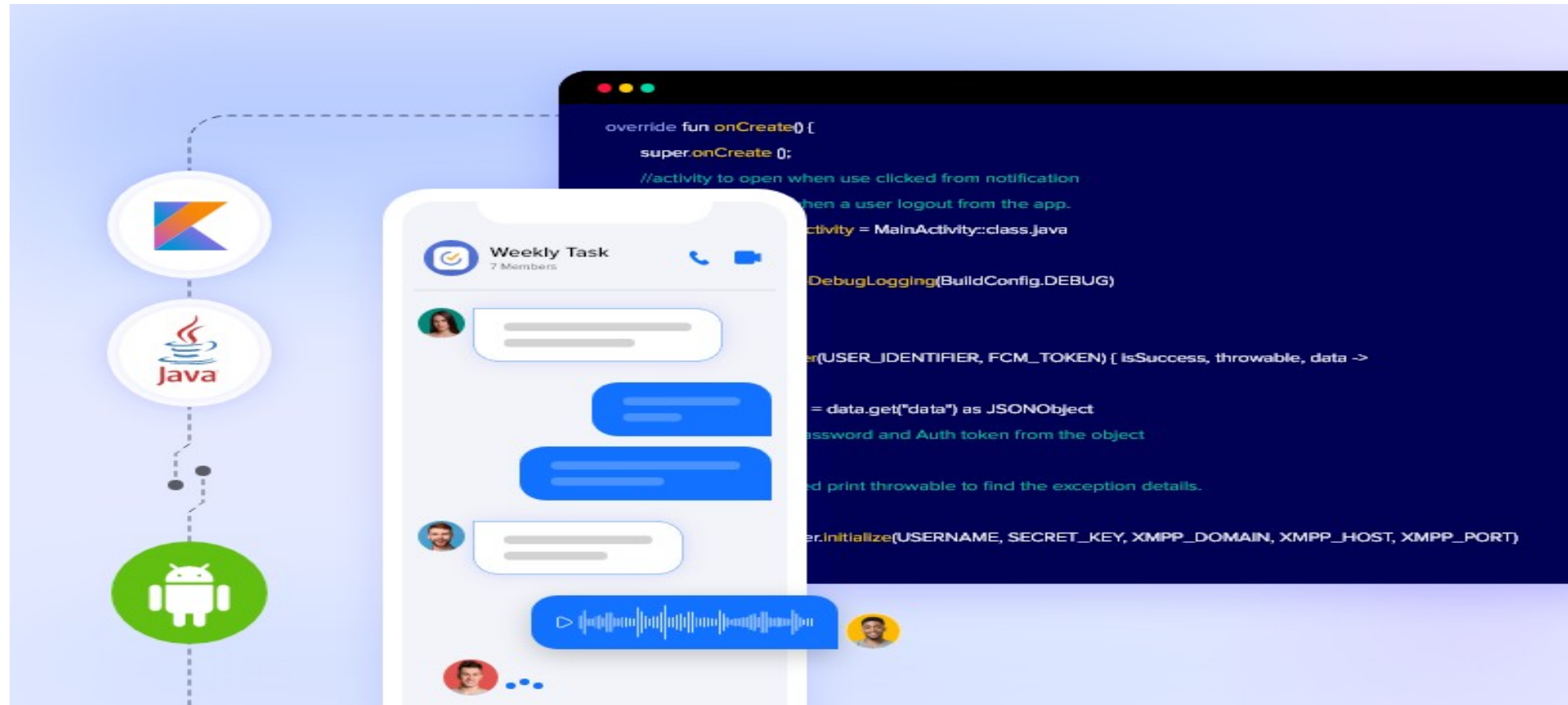
- `// Receive message`
- `socket.on('chatMessage', (msg) =>`
- `{ const item = document.createElement('li');`
- `item.textContent = msg.content; document.getElementById('messages').appendChild(item);`
- `});`

LOGOUT PROCESS

- // Logout document
- `.getElementById('logoutButton').onclick = () =>`
- `{`
- `document.getElementById('auth').style.display = 'block';`
`document.getElementById('chat').style.display = 'none';`
- `};`



SAMPLE OF REAL TIME CHAT APPLICATION



CONCLUSION

- The development of a real-time chat application using JavaScript and Socket.IO demonstrates the power of modern web technologies to facilitate instant communication.
- This project encompasses key aspects of web development, including user authentication,
- message handling, and real-time updates, while remaining accessible to developers of varying skill levels.

