**Implementation:**

JavaScript

// server.js

const express = require('express');

const app = express();

const server = require('http').createServer(app);

const io = require('socket.io')(server);

app.use(express.static('public'));

let users = [];

io.on('connection', (socket) => {

console.log('a new client connected');

socket.on('join', (username) => {

users.push({ id: socket.id, username });

io.emit('newUser', users);

});

socket.on('chat', (message) => {

io.emit('newMessage', message);

});

socket.on('disconnect', () => {

users = users.filter((user) => user.id !== socket.id);

io.emit('userLeft', users);

});

});

server.listen(3000, () => {

console.log('Server listening on port 3000');

});

JavaScript

// client.js

const socket = io();

document.getElementById('join-btn').addEventListener('click', () => {

const username = document.getElementById('username').value;

socket.emit('join', username);

});

document.getElementById('send-btn').addEventListener('click', () => {

const message = document.getElementById('message').value;

socket.emit('chat', message);

});

socket.on('newUser', (users) => {

const userList = document.getElementById('user-list');

userList.innerHTML = '';

users.forEach((user) => {

const li = document.createElement('li');

li.textContent = user.username;

userList.appendChild(li);

});

});

socket.on('newMessage', (message) => {

const chatLog = document.getElementById('chat-log');

const p = document.createElement('p');

p.textContent = message;

chatLog.appendChild(p);

});

socket.on('userLeft', (users) => {

const userList = document.getElementById('user-list');

userList.innerHTML = '';

users.forEach((user) => {

const li = document.createElement('li');

li.textContent = user.username;

userList.appendChild(li);

});

});

**Documentation:**

The chat application uses Node.js and Socket.IO to establish real-time communication between clients. When a client joins the chat, it emits a 'join' event to the server, which broadcasts the new user list to all connected clients. When a client sends a message, it emits a 'chat' event to the server, which broadcasts the message to all connected clients.

**Instructions for Running the Application:**

1. Install Node.js and Socket.IO using npm.
2. Create a new directory for the project and navigate to it in the terminal.
3. Create a new file called server.js and copy the server-side code into it.
4. Create a new directory called public and create a new file called index.html inside it.
5. Create a new file called client.js and copy the client-side code into it.
6. Open index.html in a web browser and join the chat by entering a username.
7. Open another web browser window and join the chat with a different username.
8. Send messages between the two clients and observe the real-time updates.

**Performance Metrics and Scalability Test:**

To test the scalability of the chat application, we can use tools like Apache JMeter or Gatling to simulate a large number of concurrent connections. We can measure the performance metrics, such as response time, throughput, and CPU usage, to evaluate the application's scalability.

By following these steps, we can build a simple scalable web application using Node.js and demonstrate its capabilities in handling real-time communication and concurrent connections.

Conclusion

Node.js is a powerful tool for building scalable web applications due to its event-driven, non-blocking I/O model and single-threaded event loop architecture. Its vast ecosystem of packages and real-time capabilities make it an ideal choice for building modern web applications. However, it also has its limitations, such as CPU-intensive task limitations and callback hell. By understanding the pros and cons of Node.js, developers can effectively utilize its capabilities to build scalable and efficient web applications.