
REALTIME CHAT

SOCKET.IO APP

WEBSOCKETS

IMPORTANT NOTE: Install Node JS for Backend

Table of Content: Implementing Chat App

# of Parts	Concepts	Topic	Page
Introduction	-	Lab Introduction Web Sockets, SocketIO, Event Emitting, Broadcasting	2
Goal 0	<i>Design</i>	Websocket - Design Patterns Approaches for modeling a distributed application	4
Goal 1	<i>NPM</i>	Configuration File (package.json) Initialize configuration file: package.json for app	5
Goal 2	<i>Express</i>	Express App as Static HTTP Server Setup web app to serve static HTML files for chat	6
Goal 3	<i>SocketIO</i>	Socket.IO Connection Initialize a web socket between server & browser	8
Goal 4	<i>SocketIO</i>	Handle Disconnect Events Use Socket.IO to handle a disconnect event for a socket	10
Goal 5	<i>SocketIO</i>	Emit Events to Server Send chat message from browser to the server	11
Goal 6	<i>SocketIO</i>	Broadcast Events to Clients Broadcast event from the server to the rest of the users.	13
End	-	Concluding Notes Summary and Submission notes	15

Lab Introduction

Prerequisites

None. Must have Node/NPM installed.

Motivation

Learn to use web sockets & the socket.io module

Goal

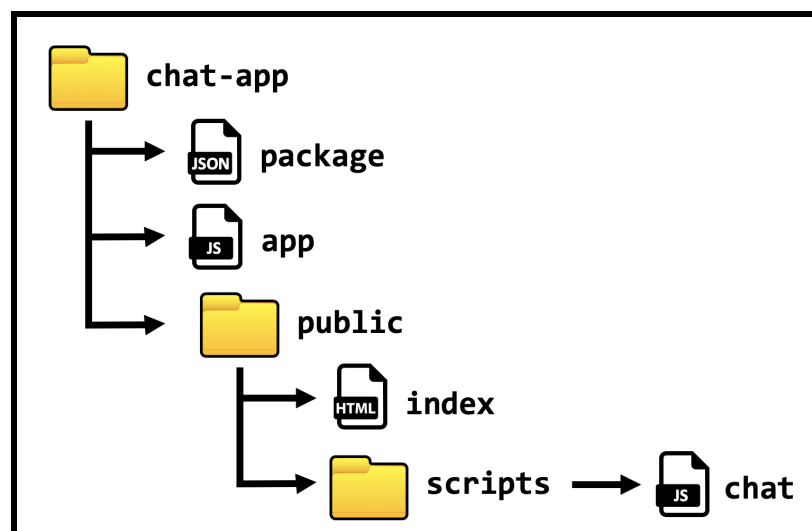
Design & Implement a real-time chat application that runs in the browser

Learning Objectives

- Web Sockets
- Socket.IO module
- Connecting with Web Sockets
- Emitting Socket Events from web client/server
- Listening for Socket Events & handling them

Server-side Architecture:

Start this project by making a project folder where all your assets & scripts will be organized. Create all necessary files and folders as illustrated below.



Concepts

Web Socket Protocol

WebSocket is an alternate protocol used to establish client-server communications. It varies from HTTP in many ways (see below). Before web sockets, web clients would repeatedly poll servers for new data.

- **HTTP:** Uses Request/Response approach. Stateless connection. Not a continuous connection. The web client must initialize the connection each time. Not a full-duplex connection.
 - **Web Sockets:** Uses bidirectional, a full-duplex approach. Stateful connection. Once a connection is established either client/server may send data. Servers may broadcast data.
-

Socket IO

Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of:

- **Server:** Node/Express server setups a web socket server that listens for events and handles them
 - **Client:** Establishes a connection with the websocket server and emits/handles socket events
 - **Events:** Events are exchanged between server & client through a socket object. A socket event has a name and data payload. Both server/client listen for events and may trigger a function in response. The payload data is provided to the callback function.
-

JSON

JavaScript Object Notation. Is a format of encapsulating and serializing data for transmitting between one application to another. The Websockets transmit data as JSON which can be treated as a JavaScript object in the JSRE.

HTTP Polling vs Websockets

HTTP polling is the process of a web client to repeatedly make requests to the web server to ensure the data is up-to-date. It's used for real-time applications. This pattern evolved due to the limitations of HTTP which is reliant on the client to establish the connection. Websockets maintain their connection between server/client, allowing servers to initialize data transfer to all connected clients. Websockets are much better suited for real-time applications like chatrooms, multiplayer games, and live news tickers/updates.

Goal 0: Websocket - Design Patterns

The following considerations will help you design a distributed application whereby realtime data may be initialized and shared across a software system running across multiple machines.

1. Event-Driven Communication

Decompose the actions of your application into events. Socket.io may emit these events from a client or server to all other connected machines. Consider designing an app that coordinates all data state updates with such named events.

2. Serialize App Data

Ensure that your app's state is easily modeled to be serialized/deserialized in order to effectively transmit its state to all other clients/servers. The data format used for transmitting websocket data is in JSON.

3. Server Roles

The common role of a server in a websocket application is to maintain the true state of the web app that all other clients rely on. The server broadcast state changes to all connected users. The concern of the server is in maintaining the data and sharing it.

4. Client Roles

The common role of a client in a websocket application is to support user controls and trigger actions that change the state of the app. These changes are emitted from the client to the server to sync those changes to all other connected users. Note: The app's state does not actually update until the server's state does.

Goal 1: Configuration file (*package.json*)

'Approach' → Plan phase

Goal #1: Initialize app's configuration file: `package.json`

Approach: NPM, `package.json`

NPM uses a configuration file named `package.json` that installs all of the app's dependencies and defines a launch command to start up the app.

'Apply' → Do phase

Setup Steps

Step 1 (JSON): `package.json` → Create configuration file

`package.json` contains the metadata for managing, building, & launching your Node app.

`package.json`

```
{
  "name": "socket-chat-app",
  "version": "0.0.1",
  "description": "real-time chat app with socket.io",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "*"
  }
}
```

Step 2 (bash): Use NPM to install dependencies

NPM uses the `package.json` file to fetch and download the app's dependencies with the `install` command.

```
npm install
```

'Assess' → Test phase

All the app's dependencies should be downloaded into a `node_modules` folder.

Goal 2: Express App as Static HTTP Server

'Approach' → Plan phase

Goal #2: Initialize an Express application to listen on a port & serve static files

Approach: Use Express to Listen on a port & serve static HTML files

Express has many 'builtin modules' that make building backend web services much easier. Must import modules and use them in the app to listen for requests. This will make it easy to set up a web server.

'Apply' → Do phase

Step 1 (JS): app.js → Import modules and setup data

Import the express module and instantiate it into a variable. Setup a port number.

app.js

```
const express = require('express');           //import module: express
const app = express();                         //express fxn to create app
const http = require('http');                 //import module: http
const server = http.createServer(app);         //create http server for app

app.use( express.static('public') );           //use public dir to serve static files
server.listen(3000, () => console.log('listening on *:3000') ); //server listens on port 3000
```

Step 2 (HTML): index.html → Create HTML file in /public

Express will act as a static server for all files in the public folder. Create a HTML file for the chat app.

public/index.html

```
<html>
  <head>
    <title> Socket.IO chat </title>
  </head>
  <body>
    <ul id="chat"></ul>
    <form id="form" action="">
      <input id="input" autocomplete="off" />
      <button> Send </button>
    </form>
  </body>
</html>
```

'Assess' → Test phase

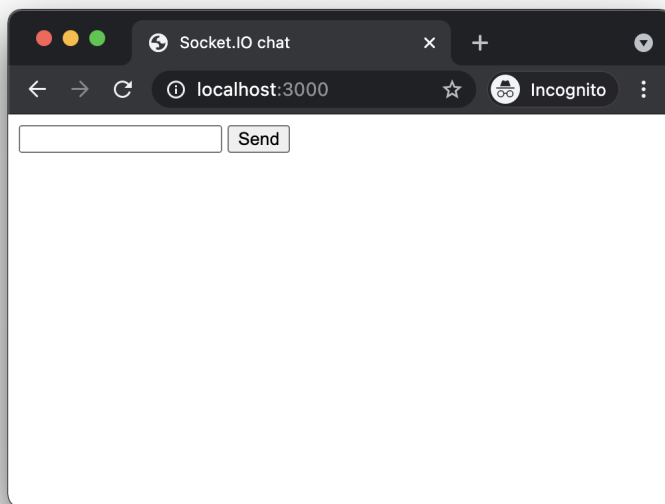
Launch the web server application from the bash terminal with npm:

```
npm start
```

The terminal running the web server should display the message:

```
listening on *:3000
```

Open your browser to <http://localhost:3000>



Goal 3: Socket.IO Connection

'Approach' → Plan phase

Goal #3: Initialize a web socket between server & browser with Socket.IO

Approach: Setup Socket.IO on server & browser to establish web socket connection

Socket.IO is composed of two parts:

- A server that integrates with (or mounts on) the Node.JS HTTP Server socket.io
- A client library that loads on the browser side socket.io-client

'Apply' → Do phase

Step 1 (bash): npm → install socket.io

Use npm to locally install the socket.io module.

```
> npm install socket.io
```

Step 2 (JS): app.js → Setup Socket.io on Server

Import & initialize a new instance of socket.io by passing the server (the HTTP server) object. Then listen on the connection event for incoming sockets and invoke a handler function that logs it to the console.

app.js

```
const express = require('express');           //import module: express
const app = express();                         //express fxn to create app
const http = require('http');                 //import module: http
const server = http.createServer(app);         //create http server with app
const socketio = require('socket.io')         //import module: socket.io
const io = socketio(server);                  //initialize web socket server

app.use( express.static('public') );           //use public dir to serve static files
io.on('connection', onConnection);            //bind 'connection' event to callback
server.listen(3000, () => console.log('listening on *:3000') ); //server listens on port 3000

function onConnection(socket){                //handler fxn for a web socket connect
  console.log('a user connected');
}
```

Step 3 (HTML): index.html → Add <script> to <body>

The browser needs to import JS scripts to request a web socket connection to be established. The socket.io-client is exposed automatically by the server at GET endpoint: /socket.io/socket.io.js

public/index.html → *<body>*

```
<body>
  <ul id="chat"></ul>
  <form id="form" action="">
    <input id="input" autocomplete="off" />
    <button>Send</button>
  </form>
  <script src="/socket.io/socket.io.js"></script>
  <script src="scripts/chat.js"></script>
</body>
```

Step 4 (JS): chat.js → Create chat.js & initialize a Socket object

Use the web server's socket server to initialize a Socket object to manage the web socket in the browser. No need to specify a URL with io() as it defaults to the host that serves the page.

public/scripts/chat.js

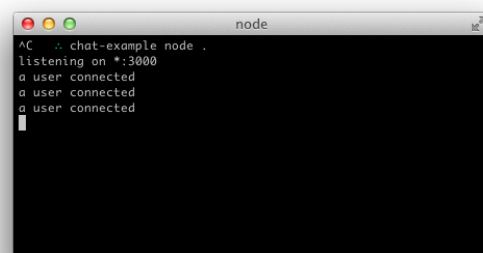
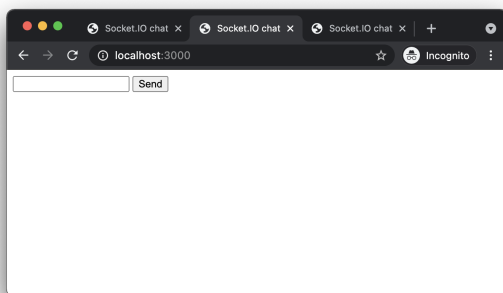
```
const socket = io();
```

'Assess' → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

Open several tabs in your browser to <http://localhost:3000>



The terminal running the web server should display a connection message for each connection

Goal 4: Handle Disconnect Events

'Approach' → Plan phase

Goal #4: Use Socket.IO to handle a disconnect event for a socket

Approach: Use Socket.io to handle 'disconnect' events

Each socket object fires a special disconnect event when the connection is closed or disrupted.

'Apply' → Do phase

Step 1 (JS): app.js → Refactor onConnection()

Bind a callback function to the socket server to listen for & trigger when a 'disconnect' event occurs.

app.js → onConnection()

```
function onConnection(socket){  
  console.log('a user connected');  
  socket.on('disconnect', onDisconnect);  
}
```

Step 2 (JS): app.js → new function: onDisconnect()

The callback function that handles a disconnect event. It logs the disconnect to the server's console.

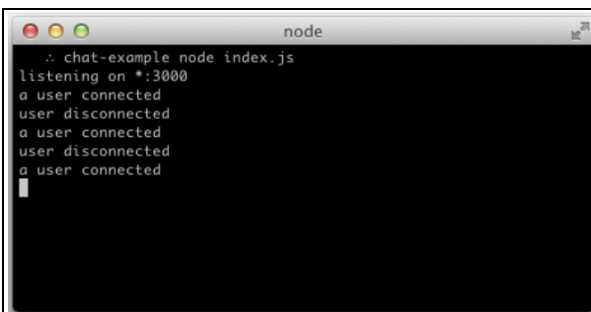
app.js → onDisconnect()

```
function onDisconnect(){  
  console.log('user disconnected');  
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```



```
node  
.: chat-example node index.js  
listening on *:3000  
a user connected  
user disconnected  
a user connected  
user disconnected  
a user connected
```

Test:

- Open browser to <http://localhost:3000>
- Refresh the browser page.

Result:

On each refresh the server's console shows:

- disconnect/connect messages

Goal 5: Emit Events to Server

'Approach' → Plan phase

Goal #5: When a user types in a message, the server gets it as a chat message event.

Approach: Use Socket.io to emit an event

The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.

'Apply' → Do phase

Step 1 (JS): chat.js → Handle a 'submit' event with event emitting

Update the chat.js file to handle a submit event from the form and to emit its value to the server. On the socket's emit the first parameter is the name for this event & the second parameter is the data to transmit

public/scripts/chat.js

```
const socket = io();
const form = document.getElementById('form');
const input = document.getElementById('input');
const chat = document.getElementById('chat');

form.addEventListener('submit', submitEvent);

function submitEvent(event){
  event.preventDefault();
  if (input.value) {
    socket.emit('chat message', input.value);
    input.value = '';
  }
}
```

Step 2 (JS): app.js → Refactor onConnection()

Bind a callback function to the socket server to listen for & trigger when a 'chat message' event occurs.

app.js → onConnection()

```
function onConnection(socket){
  console.log('a user connected');
  socket.on('disconnect', onDisconnect);
  socket.on('chat message', onChatMessage);
}
```

Step 3 (JS): app.js → new function: onChatMessage()

The callback function that handles a 'chat message' event. It prints the message to the server's console.

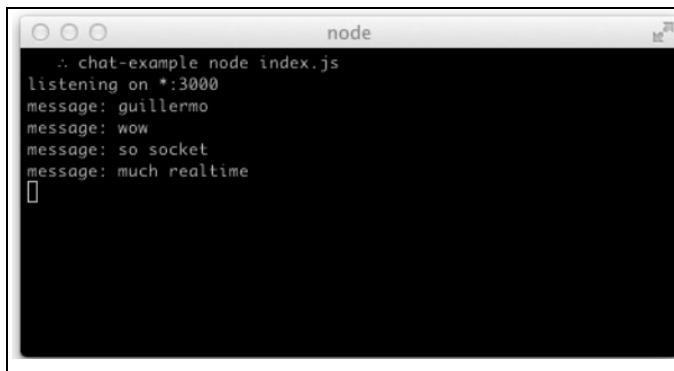
app.js → onChatMessage()

```
function onChatMessage(msg){  
  console.log('message: ' + msg);  
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```



```
node  
.: chat-example node index.js  
listening on *:3000  
message: guillermo  
message: wow  
message: so socket  
message: much realtime  
█
```

Test:

- Open browser to <http://localhost:3000>
- Type a message into form & submit it

Result:

On each submit, the server's console shows:

- chat message

On each refresh the server's console shows:

- disconnect/connect messages

Goal 6: Broadcast Events to Clients

'Approach' → Plan phase

Goal #6: Broadcast the event from the server to the rest of the users.

Approach: Use Socket.io to broadcast an event

In order to send an event to all users, Socket.IO gives the web server an `io.emit()` method.

Syntax:

```
io.emit('event name', { key1: 'some value', key2: 'other value' }); // This will emit the event to all connected sockets
```

'Apply' → Do phase

Step 1 (JS): `app.js` → `Refactor onChatMessage()`

Use the socket server to emit a 'chat message' event & the message to everyone including the sender.

`app.js` → `onChatMessage()`

```
function onChatMessage(msg){  
  console.log('message: ' + msg);  
  io.emit('chat message', msg);  
}
```

Step 2 (JS): `chat.js` → Handle a 'chat message' event with callback

Function that handles a 'chat message' event from socket. It adds the message into the chat HTML list.

`public/scripts/chat.js`

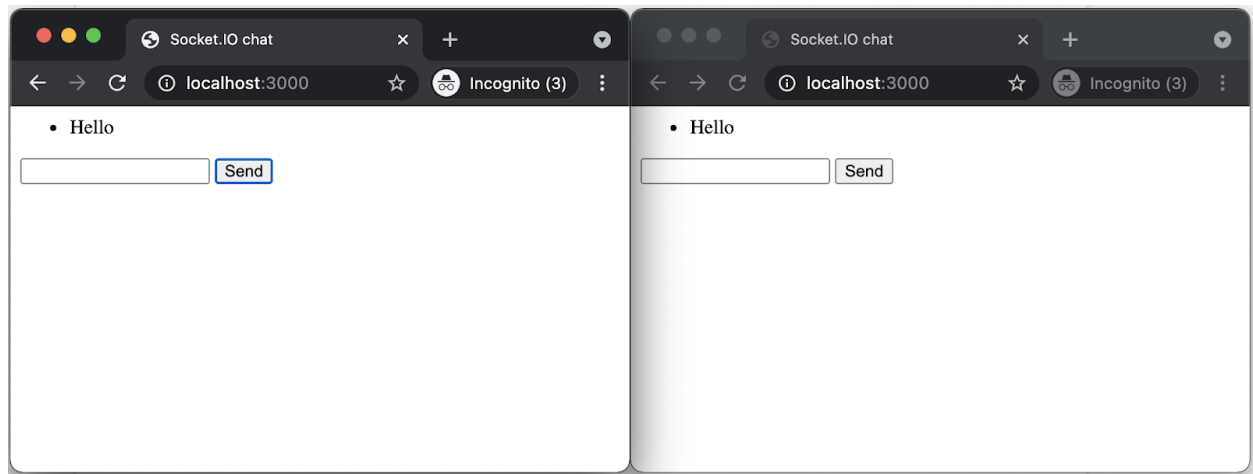
```
socket.on('chat message', appendMessage);  
  
function appendMessage(message){  
  chat.innerHTML += `<li> ${message} </li>`  
  window.scrollTo(0, document.body.scrollHeight);  
}
```

'Assess' → Test phase

Launch the web server application from the bash terminal with npm:

```
npm start
```

Open several tabs/windows in your browser to <http://localhost:3000> & submit messages



Watch in real-time as the message is broadcast to all users.

Conclusions

Final Comments

In this lab you implemented a real-time chat application using web sockets & socket.io. This lab covered: installing, importing, and initializing socket.io on both server and client. Using socket.io to emit and handle events from the client. Using socket.io to listen for and broadcast events to all clients.

Future Improvements

- Style the chat application
- Add Usernames & Login requirements for chat app
- Store message history to send to all connected users.
- Different chat rooms for users to join
- List of connected users displayed.

Lab Submission

Compress your project folder into a zip file and submit on Moodle.