**Socket.io with NodeJS**



Hello dev, in this post, we will learn about socket.io package which is used for different purpose where we wish to send and receive responses to and from server without closing our connection.

**Famous Application of Socket.io**

Here are a few examples of popular applications and use cases where Socket.IO is utilized:

***Chat Applications***  
Many chat applications, including Slack, use Socket.IO for real-time messaging between users. It allows for instant message delivery without the need for constant polling.

***Online Gaming***  
Online multiplayer games often use Socket.IO to enable real-time communication between players and the game server. It ensures that game state updates are quickly transmitted to all connected clients.

***Collaborative Editing Tools***  
Tools like Google Docs and Microsoft Office Online use real-time collaboration features powered by Socket.IO. Multiple users can edit a document simultaneously, and changes are instantly reflected for all participants.

***Financial Applications***  
Real-time stock market dashboards and financial trading platforms use Socket.IO to provide users with instant updates on stock prices, market trends, and other financial data.

***Live Sports Updates***  
Sports websites and applications use Socket.IO to deliver live updates, scores, and commentary to users in real-time during live sports events.

***Social Media Feeds***  
Some social media platforms, like Facebook and Twitter, use Socket.IO to provide users with real-time notifications, messages, and updates on their feeds.

***IoT (Internet of Things)***  
Socket.IO is used in IoT applications where devices need to communicate in real-time. It facilitates the exchange of data between IoT devices and the server.

***Collaborative Drawing Apps***  
Applications like Microsoft Whiteboard or collaborative drawing tools often use Socket.IO to allow multiple users to draw on a shared canvas simultaneously.

Now, we implement a simple chat application.

First we will create a dedicated folder and create a package.json file with following command.

npm init -y

Next, we will install packages of express and socket.io.

npm i express socket.io

We will create app.js file to implement backend.

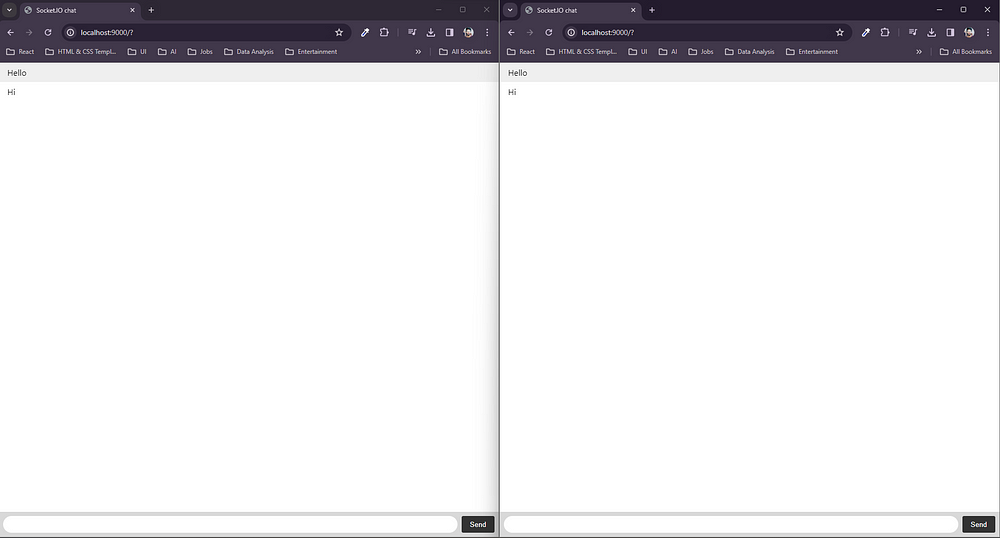
// app.js  
  
  
// Importing required modules  
const express = require('express');  
const app = express();  
const http = require('http');  
const server = http.createServer(app);  
const { Server } = require('socket.io');  
const io = new Server(server);  
  
// Setting up a route to serve the HTML file on the root path  
app.get('/', (req, res) => {  
 res.sendFile(\_\_dirname + '/index.html');  
});  
  
// Handling WebSocket connections when a client connects  
io.on('connection', socket => {  
 // Listening for 'chat message' events from clients  
 socket.on('chat message', msg => {  
 // Broadcasting the received message to all connected clients, including the sender  
 io.emit('chat message', msg);  
 });  
});  
  
// Starting the server and listening on port 9000  
server.listen(9000, () => {  
 console.log('Server running on port 9000');  
});

Now, we will develop our front-end in a file index.html

<!DOCTYPE html>  
<html>  
  
<head>  
 <title>Socket.IO chat</title>  
 <style>  
 body {  
 margin: 0;  
 padding-bottom: 3rem;  
 font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Helvetica, Arial, sans-serif;  
 }  
  
 #form {  
 background: rgba(0, 0, 0, 0.15);  
 padding: 0.25rem;  
 position: fixed;  
 bottom: 0;  
 left: 0;  
 right: 0;  
 display: flex;  
 height: 3rem;  
 box-sizing: border-box;  
 backdrop-filter: blur(10px);  
 }  
  
 #input {  
 border: none;  
 padding: 0 1rem;  
 flex-grow: 1;  
 border-radius: 2rem;  
 margin: 0.25rem;  
 }  
  
 #input:focus {  
 outline: none;  
 }  
  
 #form>button {  
 background: #333;  
 border: none;  
 padding: 0 1rem;  
 margin: 0.25rem;  
 border-radius: 3px;  
 outline: none;  
 color: #fff;  
 }  
  
 #messages {  
 list-style-type: none;  
 margin: 0;  
 padding: 0;  
 }  
  
 #messages>li {  
 padding: 0.5rem 1rem;  
 }  
  
 #messages>li:nth-child(odd) {  
 background: #efefef;  
 }  
 </style>  
</head>  
  
<body>  
 <ul id="messages"></ul>  
 <form id="form" action="">  
 <input id="input" autocomplete="off" /><button>Send</button>  
 </form>  
  
 <script src="/socket.io/socket.io.js"></script>  
 <script>  
 // Creating a socket instance to connect to the server  
 var socket = io();  
  
 var messages = document.getElementById('messages');  
 var form = document.getElementById('form');  
 var input = document.getElementById('input');  
  
 form.addEventListener('submit', function (e) {  
 e.preventDefault();   
  
 if (input.value.trim() !== '') {  
 // Emitting a 'chat message' event to the server with the input value  
 socket.emit('chat message', input.value);  
 input.value = '';  
 }  
 });  
  
 // Listening for 'chat message' events from the server  
 socket.on('chat message', function (msg) {  
 var item = document.createElement('li');  
 item.textContent = msg;  
 messages.appendChild(item);  
 window.scrollTo(0, document.body.scrollHeight);  
 });  
 </script>  
  
</body>  
  
</html>

Finally, run our project.

node app.js



And that’s it. Cool! 😃

For further application, you can visit socket.io official [website](https://socket.io/).

**Deploying secure TypeScript Node.js WebSocket server on Koyeb**

[[](https://medium.com/@goradux?source=post_page-----89c9113af508--------------------------------)](https://medium.com/@goradux?source=post_page-----89c9113af508--------------------------------)

[[Make It New](https://makeitnew.io/?source=post_page-----89c9113af508--------------------------------)](https://makeitnew.io/?source=post_page-----89c9113af508--------------------------------)

[Albert Asratyan](https://medium.com/@goradux?source=post_page-----89c9113af508--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Facb1d170d2cf&operation=register&redirect=https%3A%2F%2Fmakeitnew.io%2Fdeploying-secure-typescript-node-js-websocket-server-on-koyeb-89c9113af508&user=Albert+Asratyan&userId=acb1d170d2cf&source=post_page-acb1d170d2cf----89c9113af508---------------------post_header-----------)

Published in

[Make It New](https://makeitnew.io/?source=post_page-----89c9113af508--------------------------------)

·

7 min read

·

Jul 26

244

1

This blog post is a step-by-step tutorial on deploying a simple but secure WebSocket server using Node.js + TypeScript on [Koyeb](https://www.koyeb.com/" \t "_blank). All of the code is publicly available [here](https://github.com/Goradux/ws-minimal).

**Why deploy on Koyeb?**

In short: generous **free tier**+ **HTTPS/WSS encryption**out of the box during deployment.

[Koyeb](https://www.koyeb.com/) is a zero-configuration cloud provider that focuses on developer experience and the ability to push code to production in minutes. It also offers its users two nano-sized server instances (0.25 vCPU 256 MB RAM) free of charge. Their zero-config approach manifests itself during the deployment pipeline setup, where the only things that need to be configured are:

1. GitHub/Docker integration — connect the repository and specify the entry point;
2. Instance size and instance scaling — up to 8 vCPU per instance and up to 10 concurrent instances;
3. Environmental variables/publicly exposed ports.

Perfect for personal or small-scale projects!

**What is WebSocket?**

[WebSocket (WS)](https://datatracker.ietf.org/doc/html/rfc6455) is a communications protocol that provides a full-duplex communication channel over a single TCP connection. Websockets allow real-time data transfer, enabling the server to push information to the client without the need for the client to send requests continuously. The most common WebSocket applications are real-time, such as chats, collaborative tools, multiplayer games, and trading platforms.

Like HTTP and HTTPS, WS has a secure WSS variant that establishes an encrypted connection over TLS.

In browsers, WebSocket is implemented via the [WebSocket API](https://developer.mozilla.org/en-US/docs/Web/API/WebSocket). In Node.js, the WebSocket protocol is implemented via the **[ws](https://github.com/websockets/ws" \t "_blank)**[library](https://github.com/websockets/ws" \t "_blank).

**Prerequisites**

1. A [GitHub](http://www.github.com/) account (and basic git knowledge);
2. A [Koyeb](http://koyeb.com/" \t "_blank) account;
3. Locally installed [Node.js 18.0](https://nodejs.org/en) or newer.

**Creating a WebSocket server**

First, let’s create a new GitHub repository and initialize a new Node project with the following package.json in it:

// package.json  
{  
 "name": "sample-ws-app",  
 "version": "0.1.0",  
 "private": true,  
 "scripts": {  
 "build": "tsc ./index.ts --outDir build",  
 "start": "node ./build/index.js",  
 "start:dev": "npm run build && npm run start"  
 },  
 "dependencies": {  
 "ws": "^8.13.0"  
 },  
 "devDependencies": {  
 "@types/ws": "^8.5.4",  
 "ts-node": "10.9.1",  
 "typescript": "^5.0.2"  
 }  
}

To install all of the dependencies, run npm install in the terminal within that directory. There are a couple of commands that we are going to use:

1. build — for transpiling our TypeScript code to JavaScript. --outDir build tells TypeScript to save the .js files into /build folder;
2. start — for starting the transpiled JavaScript code;
3. start:dev — a shortcut that executes both of the commands above;

Now, let’s create an entry point to our server at index.ts :

// index.ts  
import \* as ws from "ws";  
  
const PORT = 8080;  
  
const users = new Map<string, ws.WebSocket>(); // 1.  
  
const server = new ws.WebSocketServer({ port: PORT }, () => {  
 console.log(`Server started on port ${PORT}`);  
});  
  
// 2.  
server.on("connection", function (userSocket, \_incomingMessage) {  
 const userId = String(Math.random()); // 3.  
 users.set(userId, userSocket); // 4.  
  
 // 5.  
 userSocket.on("message", function (rawMessage) {  
 const message = rawMessage.toString();  
 console.log(`Received a new message from client ${userId}: ${message}`);  
 console.log("Sending it back.");  
 userSocket.send(message);  
 console.log("Sending it to everyone.");  
 users.forEach(user => {  
 user.send(`${userId} says: ${message}`);  
 })  
 });  
  
 // 6.  
 userSocket.on("close", function (code, \_reason) {  
 users.delete(userId);  
 console.log(`${userId} closed the connection: exit code ${code}`);  
 })  
})

Let’s quickly discuss the highlighted points of interest:

1. Whenever a new user connects to our server, a new object of type ws.WebSocket gets created. It allows us to receive and send messages from/to the client, so we need to save it somewhere. Any key/value store, such as Map , is a great option, since we need to have both the identifier and the WebSocket itself;
2. We define what happens when a new client connects to the server;
3. We need to generate a userId. It can either be provided by an authenticated user or be generated server-side. Let’s just generate a random number to keep it simple;
4. We save our newly connected user to the pool of users;
5. We define what happens when a known user sends a message to the server. The incoming message is of type Buffer | ArrayBuffer | Buffer[] , so it needs to be converted to a string first. Then the given example echoes it back to the original sender and also sends it to all connected clients to show what can be done;
6. Lastly, when the connection to the client is lost (for example, if the user closed the browser), then we need to delete the closed socket from the list of active clients.

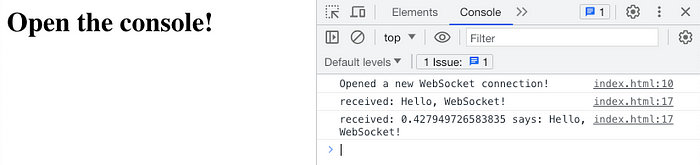
The server is done! To start it locally, run npm start:dev .

**Testing the server code locally**

The easiest way to test the server is to create a simple HTML page with a WebSocket code script. Sample index.html page:

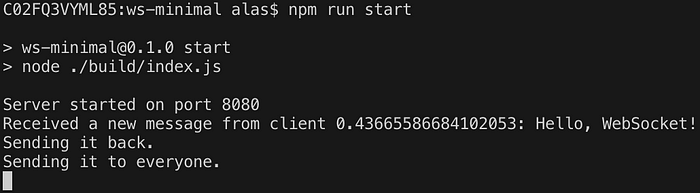
<!-- index.html -->  
<html>  
 <body>  
 <h1>Open the browser console to see the messages.</h1>  
  
 <script>  
 const PORT = 8080; // should match with the server PORT variable  
 let ws = new WebSocket(`ws://localhost:${PORT}`);  
  
 ws.onopen = function () {  
 console.log(`Opened a new WebSocket connection!`);  
 const data = "Hello, WebSocket!"  
 ws.send(data);  
 };  
  
 ws.onmessage = function (message) {  
 console.log(`received: ${message.data}`);  
 };  
 </script>  
 </body>  
</html>

Note that the browser is using the official WebSocket API, whereas our Node server is using the ws library. After opening the index.html with the server running in the background, you should see the following output in the browser:



Browser console output

And the following output in the server terminal:



Server output

It works!

As a side note, it is common to send JSON payloads instead of strings like in the example code. However, they would need to be stringified when sending (using JSON.stringify()) and parsed when receiving (using JSON.parse() ). On top of this, received data needs to be validated, but this is outside of the scope of this blog post.

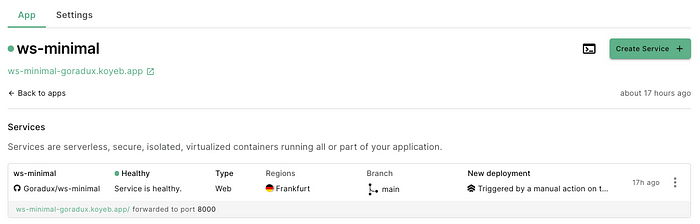
**Deploying to Koyeb**

Now that we have the working code, we need to deploy it. For the first-time setup, the easiest way is to use Koyeb website UI. When creating a new deployment (by clicking the Create App button), we will be prompted to connect to GitHub and grant access to the repositories. It is sufficient to give rights only to the current repository (in my case, ws-minimal).

After granting access and choosing the correct repository, we will be prompted to configure the deployment. We need to modify the following settings:

1. In *Build and Deployment Settings* for *Buildpack*, override the build command to npm install && npm run build and override the run command to npm run start ;
2. Keep the *Autodeploy* setting **on**. This is where the magic happens: whenever a new change is pushed to the remote, the Koyeb pipeline will trigger and redeploy the instance with the new changes;
3. Change the instance size from *Micro* to *Nano*;
4. Under *Advanced*settings, add PORT environmental variable with any numerical value (for example, 8000). Expose the port with the same value pointing to root / . Make sure it is set to *Public*.
5. Click *Deploy*.

If done correctly, the instance should spin up successfully within ~2 minutes. The running instance should look like this:



Running server instance. Note the generated entry point URL

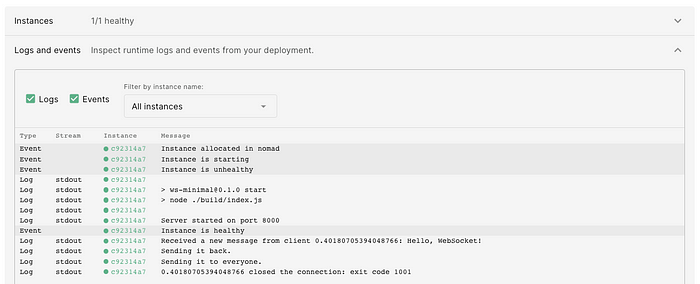
That’s it! Now the Koyeb redeployment pipeline will trigger every time a new commit to the main branch is made.

**Testing the deployed server**

The deployed server receives a custom URL generated by Koyeb automatically. In our case, it is ws-minimal-goradux.koyeb.app . This is the URL that we are going to use in our index.html instead of localhost:8080 for testing the connection. Note that the **Koyeb server redirects traffic to the exposed port automatically**, so we don’t need to specify the port in the index.html file anymore. Moreover, Koyeb provides out-of-the-box traffic encryption, so we are able to use wss:// instead of ws:// without any extra setup. Thus, we update our code with the following:

// index.html  
// old  
// let ws = new WebSocket(`ws://localhost:${PORT}`);  
// new, automatically redirects requests to port 8000  
let ws = new WebSocket(`wss://ws-minimal-goradux.koyeb.app`);

After opening the newly updated index.html file in the browser, the Koyeb console should produce the following output:



Koyeb server logs (Overview tab of the deployed service)

**Conclusion**

This tutorial shows by example how to deploy a barebones WebSocket Node.js server. It should be sufficient to be used as a blueprint for building any proper WebSocket applications.

Such a server can be deployed on any other cloud platform as well (AWS, Azure, GCP, etc.). However, then it becomes important to manually set up the TLS/SSL encryption to be able to use wss:// for secure communication. Using wss:// unlocks the ability to [safely use URL parameters for passing data to the server](https://stackoverflow.com/questions/499591/are-https-urls-encrypted). This is particularly useful for passing ID data such as session ids or session tokens that can be used for user validation when establishing the connection with the server. Koyeb’s zero-config approach provides security by default and allows developers to focus on delivering code rather than setting up infrastructure. Give it a try if you get a chance!

Make It Known!

/Albert Asratyan @ [Netlight 🦄](https://www.netlight.com/" \t "_blank)

**Working with WebSocket in Node.js using TypeScript**

[[Vitaliy Korzhenko](https://medium.com/@vitaliykorzenkoua?source=post_page-----1aebb8a06bd6--------------------------------)](https://medium.com/@vitaliykorzenkoua?source=post_page-----1aebb8a06bd6--------------------------------)

[Vitaliy Korzhenko](https://medium.com/@vitaliykorzenkoua?source=post_page-----1aebb8a06bd6--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Fa43fa1979114&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40vitaliykorzenkoua%2Fworking-with-websocket-in-node-js-using-typescript-1aebb8a06bd6&user=Vitaliy+Korzhenko&userId=a43fa1979114&source=post_page-a43fa1979114----1aebb8a06bd6---------------------post_header-----------)

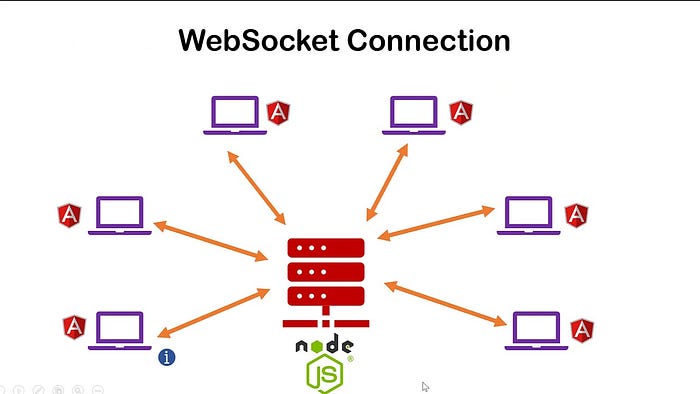
3 min read

·

Jul 8

12

1



WebSocket provides a powerful mechanism for establishing bidirectional communication channels between clients and servers. In Node.js, we can leverage the ws package along with TypeScript to conveniently implement WebSocket functionality. In this article, we will explore how to work with WebSocket in Node.js using TypeScript and provide examples to demonstrate their usage.

**Installation**

To get started, we need to install the ws package and TypeScript typings using npm:

$ npm install ws  
$ npm install --save-dev @types/ws

**Creating a WebSocket Server**

Let’s begin by creating a WebSocket server in Node.js using TypeScript. Save the following code in a file named server.ts

import WebSocket from 'ws';  
  
const wss = new WebSocket.Server({ port: 8080 });  
  
wss.on('connection', (ws: WebSocket) => {  
 console.log('New client connected');  
  
 ws.on('message', (message: string) => {  
 console.log(`Received message: ${message}`);  
 ws.send(`Server received your message: ${message}`);  
 });  
  
 ws.on('close', () => {  
 console.log('Client disconnected');  
 });  
});

In this code, we import the WebSocket type from the ws package and create a WebSocket server that listens on port 8080. When a client connects, a "connection" event is emitted, and we log a message to the console. When a message is received from the client, we log it and send a response back to the client.

**Creating a WebSocket Client**

Now let’s create a WebSocket client using TypeScript to connect to the server we just created. Save the following code in a file named client.ts:

import WebSocket from 'ws';  
  
const ws = new WebSocket('ws://localhost:8080');  
  
ws.on('open', () => {  
 console.log('Connected to server');  
  
 ws.send('Hello, server!');  
});  
  
ws.on('message', (message: string) => {  
 console.log(`Received message from server: ${message}`);  
});  
  
ws.on('close', () => {  
 console.log('Disconnected from server');  
});

In this code, we import the WebSocket type from the ws package and create a WebSocket connection to the server at ws://localhost:8080. After the connection is established, an "open" event is emitted, and we log a message to the console. We then send a message to the server. When a message is received from the server, we log it.

**Broadcasting Messages to All Clients**

WebSocket allows us to broadcast messages to multiple clients. Let’s modify our server code to broadcast messages to all connected clients. Update the server.ts file with the following code:

import WebSocket from 'ws';  
  
const wss = new WebSocket.Server({ port: 8080 });  
  
wss.on('connection', (ws: WebSocket) => {  
 console.log('New client connected');  
  
 ws.on('message', (message: string) => {  
 console.log(`Received message: ${message}`);  
 wss.clients.forEach((client) => {  
 client.send(`Server received your message: ${message}`);  
 });  
 });  
  
 ws.on('close', () => {  
 console.log('Client disconnected');  
 });  
});

In this modified code, when a message is received from a client, we iterate over all connected clients using the wss.clients property and send the message to each one.

WebSocket in Node.js using TypeScript provides a powerful mechanism for real-time communication between clients and servers. In this article, we explored how to work with WebSocket in Node.js using the ws package with TypeScript typings. We covered the basics of creating a WebSocket server, connecting a client to the server, and broadcasting messages to all connected clients. Feel free to explore further and leverage the full potential of WebSocket in your Node.js applications.