



**¡Les damos la
bienvenida!**

¿Comenzamos?

Esta clase va a ser

- grabada

Unidad 6. DESARROLLO AVANZADO DE BACKEND

Desarrollo de Aplicaciones en Tiempo Real con Websockets

Objetivos de la clase

- Comprender la diferencia entre HTTP y Websockets
- Configurar un servidor websocket dentro de nuestro servidor express.
- Integrar websocket en nuestras plantillas

Objetivos de la clase

- Desarrollar una aplicación chat con websockets
- Comprender el uso de Sweetalert como sistema de autenticación intermedio.
- Hacer Deploy de nuestra primera aplicación utilizando Glitch.com

Objetivos de la clase

- Desarrollar una aplicación chat con websockets
- Comprender el uso de Sweetalert como sistema de autenticación intermedio.
- Hacer Deploy de nuestra primera aplicación utilizando Glitch.com

Glosario

Plantilla: Documento HTML con marcas reemplazables para poder ser reemplazadas por un motor de plantillas.

Motor de plantilla: Librería desarrollada para tomar un HTML y reemplazar datos en éste para generar un efecto de dinamismo en la página.

Página estática: Página que no requiere cambio de elementos ni interacción compleja con el usuario.

Handlebars: Motor de plantillas basado en la marca `{{}}`

Layout: Marco de disposición que requiere handlebars para poder encapsular en un solo cuerpo HTML todas las vistas de diferentes plantillas

express-handlebars: paquete o módulo de npm para conectar handlebars con express



Para pensar

Ya analizamos un caso práctico donde el uso de websockets es necesario.

¿Qué otros casos requerirían de websocket?

Contesta la encuesta de Zoom

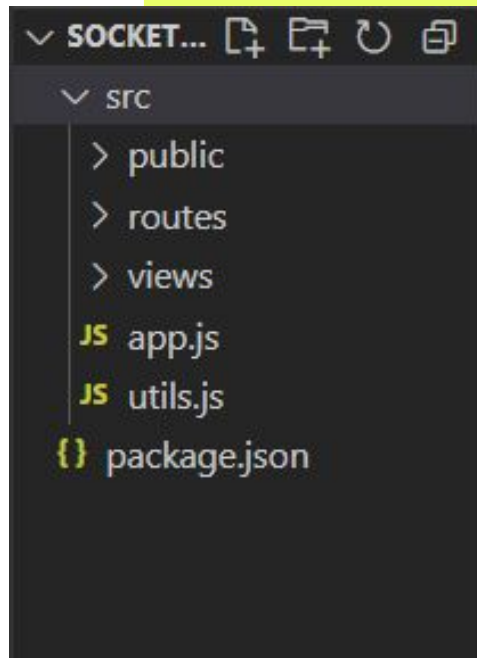
Instalación y configuración de Socket.io

1. Tener listo un servidor Express

Para poder trabajar con websockets en Express, necesitamos un servidor para que trabajen en conjunto, de manera que levantaremos un servidor express como ya lo conocemos.

Utilizaremos la misma estructura de plantillas trabajadas con Handlebars, de manera que debemos contar con la arquitectura

La estructura inicial deberá ser como lo indica la imagen





2. Realizar las instalaciones

Una vez que tenemos la estructura de carpetas inicial, realizamos la instalación de nuestros elementos cruciales para trabajar con websockets.

- ✓ **express:** Nuestro servidor principal.
- ✓ **express-handlebars:** Para las plantillas donde colocaremos el socket de lado de cliente.
- ✓ **socket.io:** Para trabajar con websockets, tanto para cliente como para servidor.

```
\Socketioexample> npm install express express-handlebars socket.io
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io : **app.js**

```
JS app.js  ×
src > JS app.js > ...
1 import express from 'express';
2 import __dirname from './utils.js';
3 import handlebars from 'express-handlebars';
4 import viewsRouter from './routes/views.router.js'
5 import {Server} from 'socket.io'; //Este import es nuevo, este "Server" se creará a partir del server HTTP
6
7 const app = express();
8 const httpServer = app.listen(8080,()=>console.log("Listening on PORT 8080")); //Sólo el Server HTTP
9
10 //¡Ahora algo nuevo! Creamos un servidor para sockets viviendo dentro de nuestro servidor principal
11 const socketServer = new Server(httpServer); //socketServer será un servidor para trabajar con sockets
12 //Configuramos todo lo referente a plantillas.
13 app.engine('handlebars',handlebars.engine());
14 app.set('views',__dirname+'/views');
15 app.set('view engine','handlebars');
16 app.use(express.static(__dirname+'/public')); //Recuerda que es importante para tener archivos js y css en plantillas
17 app.use('/',viewsRouter);
18
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io : **utils.js**

(solo válido si trabajamos con type: module)

```
JS app.js  JS utils.js  X
src > JS utils.js > [e] default
1  import {fileURLToPath} from 'url';
2  import { dirname } from 'path';
3  const __filename = fileURLToPath(import.meta.url);
4  const __dirname = dirname(__filename);
5
6  export default __dirname;
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io: **views.router.js**

```
src > routes > JS views.router.js > [e] default
```

```
1 import express from 'express';
2
3 const router = express.Router();
4
5 ∨ router.get('/', (req, res) => {
6   |   res.render('index', {}); //De momento sólo renderizaremos la vista, no pasaremos objeto.
7   | })
8
9 export default router;
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io: **index.handlebars**

index.handlebars X

src > views > index.handlebars > div

```
1 <div>
2   Todo listo para trabajar con websockets
3 </div>
```



APROXIMACIÓN AL PROCESO

4. Agregar js a la carpeta public y a nuestro index.handlebars

```
EXPLORER
...
index.handlebars x JS index.js

OPEN EDITORS
x index.handlebar...
JS index.js src\publi...

SOCKETIOEXAMPLE
> node_modules
> src
  > public\js
    JS index.js
  > routes
  > views
    > layouts
    index.handlebars
  JS app.js
  JS utils.js
  {} package-lock.json
  {} package.json

src > views > index.handlebars > script
1 <div>
2   Todo listo para trabajar con websockets
3 </div>
4 <script src="/socket.io/socket.io.js"></script>
5 <script src="/js/index.js"></script>
```

Creamos un index.js en la carpeta public/js/ y la referenciamos en nuestro index.handlebars (línea 5)

Como comentamos, el cliente también necesita instanciar su socket, entonces lo colocamos en un script con la sintaxis indicada en la imagen. (línea 4)

El script de socket siempre debe estar antes que el script propio



APROXIMACIÓN AL PROCESO

5. Levantar nuestro socket del lado del cliente en index.js

Una vez que hemos importado socket.io desde nuestro script de lado del cliente, podemos probar utilizándolo en nuestro archivo index.js.

En este archivo index.js es donde tendremos el socket/cliente para conectar con socket/servidor

JS index.js X

```
src > public > js > JS index.js > ...
```

```
1  const socket = io();
2  /**
3   * io hace referencia a "socket.io", se le llama así por convención.
4   * La línea 1 permite instanciar el socket y guardarlo en la constante "socket"
5   * Dicho socket es el que utilizaremos para poder comunicarnos con el socket del servidor.
6   * (Recuerda que, en este punto somos "clientes", porque representamos una vista)
7   */
```

Configuración y uso del socket del lado del servidor

El cliente y el servidor deben estar ligados

Hasta este momento, tenemos a socket.io de lado del cliente listo para conectarse a nuestro servidor, sin embargo, aún no hemos enseñado a nuestro servidor a escuchar el handshake por parte del cliente, para ello tenemos nuestro socketServer (normalmente se llama "io", pero lo utilizaremos como socketServer para que sea más claro).

Tomaremos nuestro socketServer y lo pondremos "a escuchar una conexión". Una vez que un socket se conecte, podemos reconocerlo y tomar alguna acción a partir de ello.

Utilizando socketServer.on para escuchar la conexión de un nuevo socket

```
//¡Ahora algo nuevo! Creamos un servidor para sockets viviendo dentro de nuestro servidor principal
const socketServer = new Server(httpServer); //socketServer será un servidor para trabajar con sockets
//Configuramos todo lo referente a plantillas.
app.engine('handlebars',handlebars.engine());
app.set('views',__dirname+'/views');
app.set('view engine','handlebars');
app.use(express.static(__dirname+'/public'));//Recuerda que es importante para tener archivos js y css en plantillas
app.use('/',viewsRouter);

socketServer.on('connection',socket=>{
  console.log("Nuevo cliente conectado")
})
```

Probando conexión cliente → servidor

Todo listo para trabajar con websockets



```
19 socketServer.on('connection', socket=>{  
20   console.log("Nuevo cliente conectado")  
21 })  
22
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Nuevo cliente conectado

¿Qué acaba de pasar?

El cliente se conecta con su websocket al socketServer (socketServer.on significa que está escuchando porque algo pase), entonces, cuando socketServer escucha que hay una nueva conexión (**connection**), muestra en consola el mensaje "Nuevo cliente conectado". Es por eso que aparece el mensaje en la consola del Visual Studio Code.

Recibiendo nuevos eventos en socket/servidor

Esta vez, una vez que el socket se ha conectado, podemos escuchar eventos de dicho socket, a partir de la sintaxis indicada:

socket.on("nombre_del_evento_a_escuchar",callback con la data que me hayan enviado);

Este "evento a escuchar" tiene un identificador que el cliente tiene que colocar de su lado para poder enviar información. Podemos tener múltiples **socket.on**, para tener así escuchar diferentes eventos.

```
socketServer.on('connection',socket=>{  
  console.log("Nuevo cliente conectado")  
  //socket.on("message",data=>) significa: "escuchar cuando el socket conectado envíe un evento de tipo 'message'"  
  // En cuanto se reciba un evento de tipo "message", con la 'data' que se envió, mostrarla por consola.  
  socket.on('message',data=>{  
    console.log(data);  
  })  
})
```

Enviando información desde el cliente hacia el servidor con “emit”

Ahora, el servidor ya está escuchando por un evento con ID “message”, pero no hemos enseñado a nuestro cliente a comunicarse con el servidor.

Para enviar un mensaje desde el cliente hacia el servidor (o desde el servidor al cliente, recuerda que es bidireccional), utilizamos la palabra **emit**, el cual debe contar con el ID del mensaje a enviar, seguido del contenido de dicho mensaje.

Los IDs deben coincidir para que el mensaje llegue correctamente

Finalmente, en nuestro cliente en index.js escribiremos nuestro primer emit:

```
src > public > js > JS index.js > ...  
1  const socket = io();  
2  socket.emit('message', '¡Hola, me estoy comunicando desde un websocket!')
```

Al momento de integrar el listener con socket.on y el envío con socket.emit, debemos visualizar nuestra primera comunicación

Servidor escucha

```
socket.on('message', data=>{  
  console.log(data);  
})
```

Cliente envía

```
JS index.js  X  
src > public > js > JS index.js > ...  
1  const socket = io();  
2  socket.emit('message', '¡
```

```
[nodemon] starting `node app.js`  
Listening on PORT 8080  
Nuevo cliente conectado  
¡Hola, me estoy comunicando desde un websocket!
```

El servidor recibe el mensaje y lo muestra en consola

Enviando información desde el servidor hacia el cliente

Como comentamos, el carácter de un websocket debe ser bidireccional, eso significa que el servidor también debe poder enviar mensajes al cliente. Sin embargo, el servidor lo puede hacer de tres formas:

```
socket.on('message',data=>{  
  console.log(data);  
})  
  
socket.emit('evento_para_socket_individual','Este mensaje sólo lo debe recibir el socket');  
  
socket.broadcast.emit('evento_para_todos_menos_el_socket_actual','Este evento lo verán  
Todos los sockets conectados, MENOS el socket actual desde el que se envió el mensaje`')  
  
socketServer.emit('evento_para_todos','Este mensaje lo reciben todos los sockets conectados')  
})
```

Configurando al cliente en index.js para escuchar los eventos que envía el servidor

```
1  const socket = io();
2  socket.emit('message','¡Hola, me estoy comunicando desde un websocket!')
3
4  socket.on('evento_para_socket_individual',data=>{
5    |   console.log(data);
6  | })
7
8  socket.on('evento_para_todos_menos_el_socket_actual',data=>{
9    |   console.log(data);
10 | })
11
12 socket.on('evento_para_todos',data=>{
13 |   console.log(data);
14 | })
```



Para pensar

El cliente puede comunicarse de una sola forma con el servidor. Sin embargo, como vemos, hay diferentes formas de comunicarse desde el servidor hacia el cliente.

¿Cuándo deberíamos utilizar cada emit?

Contesta mediante el chat de Zoom



Servidor con Websocket

Aplicación más clara sobre comunicación entre sockets

Duración: 10/15 min



ACTIVIDAD EN CLASE

Servidor con Websockets

Sobre el proyecto de websocket que venimos desarrollando:

- ✓ Sobre la estructura anteriormente creada, agregar en la vista de cliente un elemento de entrada de texto donde al introducir texto, el mensaje se vea reflejado en todos los clientes conectados en un párrafo por debajo del input.
El texto debe ser enviado caracter a caracter y debe reemplazar el mensaje previo.



ACTIVIDAD EN CLASE

Servidor con Websockets

Sobre el proyecto de websocket que venimos desarrollando:

- ✓ Basado en el ejercicio que venimos realizando, ahora los mensajes enviados por los clientes deberán ser almacenados en el servidor y reflejados por debajo del elemento de entrada de texto cada vez que el usuario haga un envío. La estructura de almacenamiento será un array de objetos, donde cada objeto tendrá la siguiente estructura:
`{ socketid: (el socket.id del que envió el mensaje), mensaje: (texto enviado) }`
- Cada cliente que se conecte recibirá la lista de mensajes completa.
- Modificar el elemento de entrada en el cliente para que disponga de un botón de envío de mensaje.
- Cada mensaje de cliente se representará en un renglón aparte, anteponiendo el socket id.

Aplicación chat con Websockets

Aplicación chat con websocket

Como aprendimos la clase pasada, las aplicaciones de websocket son bastante amplias. Una de las mejores formas de comprender su aplicación, es realizando un chat comunitario.

Nuestro chat comunitario contará con:

- ✓ Una vista que cuente con un formulario para poder identificarse. El usuario podrá elegir el nombre de usuario con el cual aparecerá en el chat.
- ✓ Un cuadro de input sobre el cual el usuario podrá escribir el mensaje.
- ✓ Un panel donde todos los usuarios conectados podrán visualizar los mensajes en tiempo real
- ✓ Una vez desarrollada esta aplicación, subiremos nuestro código a glitch.com, para que todos puedan utilizarlo.

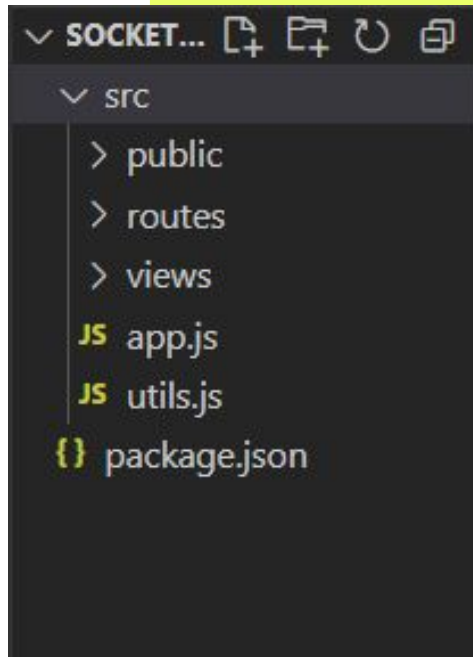
**Comencemos el proyecto
from scratch**

1. Tener listo un servidor Express

Para poder trabajar con websockets en Express, necesitamos un servidor para que trabajen en conjunto, de manera que levantaremos un servidor express como ya lo conocemos.

Utilizaremos la misma estructura de plantillas trabajadas con Handlebars, de manera que debemos contar con la arquitectura

La estructura inicial deberá ser como lo indica la imagen





APROXIMACIÓN AL PROCESO

2. Realizar las instalaciones

Una vez que tenemos la estructura de carpetas inicial, realizamos la instalación de nuestros elementos cruciales para trabajar con websockets.

- ✓ **express:** Nuestro servidor principal.
- ✓ **express-handlebars:** Para las plantillas donde colocaremos el socket de lado de cliente.
- ✓ **socket.io:** Para trabajar con websockets, tanto para cliente como para servidor.

```
\Socketioexample> npm install express express-handlebars socket.io
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io : **app.js**

```
JS app.js  X
src > JS app.js > ...
1  import express from 'express';
2  import __dirname from './utils.js';
3  import handlebars from 'express-handlebars';
4  import viewsRouter from './routes/views.router.js'
5  import {Server} from 'socket.io'; //Recuerda que este {Server} es propio de websockets.
6
7  const app = express();
8  const httpServer = app.listen(8080,()=>console.log("Listening on PORT 8080"));
9
10 const io = new Server(httpServer); //io será un servidor para trabajar con sockets, ¿por qué ahora lo llamamos "io"?
11 //Configuramos todo lo referente a plantillas.
12 app.engine('handlebars',handlebars.engine());
13 app.set('views',__dirname+'/views');
14 app.set('view engine','handlebars');
15 app.use(express.static(__dirname+'/public'));//Recuerda que es importante para tener archivos js y css en plantillas
16 app.use('/',viewsRouter);
```

¡Importante!

Seguramente estás pensando: “¿Por qué ahora `socketServer` se llama `'io'`?”

El nombre, como sabrás, no influye en nada, sólo **queremos mostrarte el nombre con el que se recomienda nombrarlo “por convención”**. Es decir, queremos que te acostumbres al nombre que encontrarás en otros proyectos.



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io : **utils.js**

(solo válido si trabajamos con type: module)

```
JS app.js    JS utils.js    X
src > JS utils.js > [e] default
1  import {fileURLToPath} from 'url';
2  import { dirname } from 'path';
3  const __filename = fileURLToPath(import.meta.url);
4  const __dirname = dirname(__filename);
5
6  export default __dirname;
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io:

views router.js

src > routes > JS views.router.js > [e] default

```
1  import express from 'express';
2
3  const router = express.Router();
4
5  router.get('/', (req, res) => {
6    res.render('index', {}); //De momento sólo renderizaremos la vista, no pasaremos objeto.
7  })
8
9  export default router;
```



APROXIMACIÓN AL PROCESO

3. Configurando nuestro servidor express con Handlebars + socket.io: **index.handlebars**

index.handlebars X

src > views > index.handlebars > div

```
1 <div>
2   Todo listo para trabajar con websockets
3 </div>
```




APROXIMACIÓN AL PROCESO

4. Agregar js a la carpeta public y a nuestro index.handlebars

```
EXPLORER
...
index.handlebars x JS index.js
OPEN EDITORS
x index.handlebar...
JS index.js src\publi...
SOCKETIOEXAMPLE
> node_modules
> src
  > public\js
    JS index.js
  > routes
  > views
    > layouts
    index.handlebars
  JS app.js
  JS utils.js
  {} package-lock.json
  {} package.json

src > views > index.handlebars > script
1 <div>
2   Todo listo para trabajar con websockets
3 </div>
4 <script src="/socket.io/socket.io.js"></script>
5 <script src="/js/index.js"></script>
```

Creamos un index.js en la carpeta public/js/ y la referenciamos en nuestro index.handlebars (línea 5)

Como comentamos, el cliente también necesita instanciar su socket, entonces lo colocamos en un script con la sintaxis indicada en la imagen. (línea 4)

El script de socket siempre debe estar antes que el script propio



APROXIMACIÓN AL PROCESO

5. Levantar nuestro socket del lado del cliente en index.js

Una vez que hemos importado socket.io desde nuestro script de lado del cliente, podemos probar utilizándolo en nuestro archivo index.js.

En este archivo index.js es donde tendremos el socket/cliente para conectar con socket/servidor

JS index.js X

src > public > js > JS index.js > ...

```
1  const socket = io();
2  /**
3   * io hace referencia a "socket.io", se le llama así por convención.
4   * La línea 1 permite instanciar el socket y guardarlo en la constante "socket"
5   * Dicho socket es el que utilizaremos para poder comunicarnos con el socket del servidor.
6   * (Recuerda que, en este punto somos "clientes", porque representamos una vista)
7   */
```

Comenzamos con las configuraciones: Sweetalert2



APROXIMACIÓN AL PROCESO

sweetalert2

Sweetalert2 nos permitirá utilizar alertas más estéticas y con más funcionalidades. En este caso la utilizaremos para dos cosas particulares:

- ✓ Para bloquear la pantalla del chat hasta que el usuario se identifique
- ✓ Para notificar a los usuarios cuando alguien se conecte al chat



APROXIMACIÓN AL PROCESO

1. Instalamos Sweetalert en la nuestra vista "index.handlebars"

```
src > views > index.handlebars > script
1  <div>
2    Todo listo para trabajar con websockets
3  </div>
4  <p id="message"></p>
5  <script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script>
6  <script src="/socket.io/socket.io.js"></script>
7  <script src="/js/index.js"></script>
```

```
src > public > js > index.js > text
1  const socket = io();
2  Swal.fire({
3    title:"Hola, Coders",
4    text:"Alerta básica con Sweetalert2",
5    icon:"success"
6  });
```

2. Utilizamos el objeto "Swal" en nuestro index.js

Resultado: Una alerta presentable, profesional, y lista para customizarse

Todo listo para trabajar con websockets



Hola, Coders

Alerta básica con Sweetalert2

OK

Vista de autenticación con Sweetalert2



APROXIMACIÓN AL PROCESO

Cambiando nuestro index.handlebars

Configuramos nuestro archivo index.handlebars para que esta vez no nos salude solamente. Colocaremos un mensaje de bienvenida, pero colocaremos dos etiquetas adicionales.

```
src > views > index.handlebars > div > p#messageLogs
1  <h1>¡Bienvenidos al CoderChat comunitario!</h1>
2  <div>
3  |   <input id="chatBox"><!-- chatBox será la caja para poder escribir nuestro mensaje-->
4  | </div>
5  <div>
6  |   <p id="messageLogs"></p> <!--message log servirá para ir colocando los mensajes que vayan llegando-->
7  | </div>
8  <script src="//cdn.jsdelivr.net/npm/sweetalert2@11"></script>
9  <script src="/socket.io/socket.io.js"></script>
10 <script src="/js/index.js"></script>
```




APROXIMACIÓN AL PROCESO

Reestructuramos nuestro index.js para un sistema de autenticación con Sweetalert2

```
src > public > js > JS index.js > ...
1  const socket = io();
2  let user; // Este "user" será con el que el cliente se identificará para saber quién escribió el mensaje.
3  let chatBox = document.getElementById('chatBox'); //Obtenemos la referencia del cuadro donde se escribirá.
4
5  Swal.fire({
6    title:"Identificate",
7    input:"text",//Indicamos que el cliente necesita escribir un texto para poder avanzar de esa alerta.
8    text:"Ingresa el usuario para identificarte en el chat",
9    inputValidator: (value) => {
10      return !value && '¡Necesitas escribir un nombre de usuario para continuar!'
11      //Esta validación ocurre si el usuario decide dar en "continuar" sin haber colocado un nombre de usuario.
12    },
13    allowOutsideClick:false // Impide que el usuario salga de la alerta al dar "click" fuera de la alerta.
14  }).then(result=>{
15    user=result.value
16    //Una vez que el usuario se identifica, lo asignamos a la variable user.
17  });
```



APROXIMACIÓN AL PROCESO

Vista si no me quiero identificar

¡Bienvenidos al CoderChat comunitario!

Identificate

Ingresá el usuario para identificarte en el chat

¡Necesitas escribir un nombre de usuario para continuar!

OK



APROXIMACIÓN AL PROCESO

Agregamos a nuestro chatBox el evento de socket

```
JS app.js JS index.js X
src > public > js > JS index.js > chatBox.addEventListener('keyup') callback
9     inputValidator: (value) => {
10         return !value && '¡Necesitas escribir un nombre de usuario para continuar!'
11         //Esta validación ocurre si el usuario decide dar en "continuar" sin haber colocado un nombre de usuario.
12     },
13     allowOutsideClick:false // Impide que el usuario salga de la alerta al dar "click" fuera de la alerta.
14 }).then(result=>{
15     user=result.value
16     //Una vez que el usuario se identifica, lo asignamos a la variable user.
17 });
18
19 chatBox.addEventListener('keyup',evt=>{
20     if(evt.key==="Enter"){ //El mensaje se enviará cuando el usuario apriete "Enter" en la caja de chat
21         if(chatBox.value.trim().length>0){ //Corroboramos que el mensaje no esté vacío o sólo contenga espacios.
22             socket.emit("message",{user:user,message:chatBox.value}); //Emitimos nuestro primer evento.
23             chatBox.value="";
24         }
25     }
26 })
```



APROXIMACIÓN AL PROCESO

Configuramos nuestro app.js para escuchar el evento "app.js"

Recibimos el evento "message", guardamos el mensaje en el arreglo, e **instantáneamente** reenviamos a todos los usuarios conectados el mensaje

```
let messages = []; //Los mensajes se almacenarán aquí
io.on('connection',socket=>{
  console.log("Nuevo cliente conectado")

  socket.on('message',data =>{//Nota cómo escucha al evento con el mismo nombre que el emit del cliente: "messages"
    messages.push(data) //Guardamos el objeto en la "base";
    io.emit('messageLogs',messages) //Reenviamos instantáneamente los logs actualizados.
    //(Nota que el evento "messageLogs" no está programado de lado del cliente, lo agregaremos más adelante)
  })
})
```



APROXIMACIÓN AL PROCESO

Configuramos nuevamente index.js para agregar el Listener del evento “messageLogs” y mostrarlo en la página

Colocamos en nuestro mismo index.js un “listener” para escuchar el evento del messageLogs y lo mostramos en la etiqueta que referenciamos para colocar logs

```
/*SOCKET LISTENERS */  
socket.on('messageLogs', data=>{  
  let log = document.getElementById('messageLogs');  
  let messages = "";  
  data.forEach(message=>{  
    messages = messages+`${message.user} dice: ${message.message}<br>`  
  })  
  log.innerHTML = messages;  
})
```

Resultado probando el chat con tres diferentes navegadores

¡Bienvenidos al CoderChat comunitario!

Mauricio dice: ¡Hola!

Dau dice: ¡Estamos chateando!

Mary dice: ¡Qué hay, chicos!



Chat websocket ampliado

Agregando últimas características a nuestro chat con
websocket

Duración: 10min



ACTIVIDAD EN CLASE

Chat websocket ampliado

Con base en el servidor con chat de websocket que se ha desarrollado. Crear nuevos eventos para que:

- ✓ Cuando el usuario se autentique correctamente, el servidor le mande los logs de todo el chat.
- ✓ Cuando el usuario se autentique correctamente, todos los demás usuarios (menos el que se acaba de registrar) reciban una notificación indicando qué usuario se acaba de conectar. (utiliza Swal toast).

```
Swal.fire({  
  text:"Nuevo usuario conectado",  
  toast:true,  
  position:"top-right"  
})
```

Apoyo: Así es como se debe ver la estructura genérica de un Toast.

Proceso de deploy de nuestra aplicación

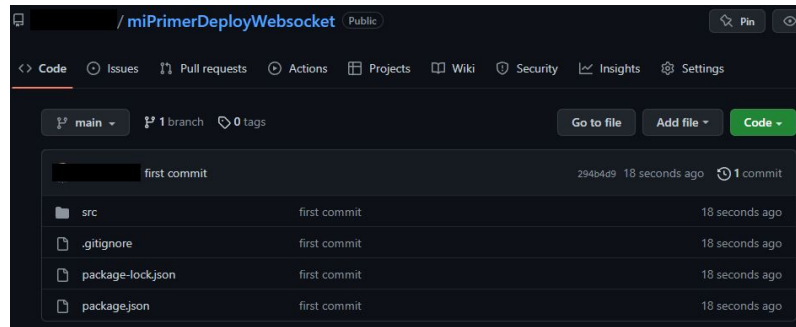


APROXIMACIÓN AL PROCESO

1. Subir nuestro código a un repositorio de Github

Glitch.com, así como cualquier otro sistema decente de deploy, es muy amigable con gestores de repositorios, de manera que la mejor forma de entregar a Glitch.com nuestro proyecto, será si éste ya vive en un repositorio.

- ✓ El repositorio debe ser público para que Glitch.com pueda acceder a éste.
- ✓ Recuerda no subir los node_modules
- ✓ Recuerda incluir el script "start" en package.json





APROXIMACIÓN AL PROCESO

2. Crear una cuenta en Glitch.com

El proceso es bastante sencillo, una vez que hayamos finalizado el proceso de registro y tengamos lista nuestra cuenta de glitch.com, visualizaremos el siguiente panel:

[Dashboard](#)[Discover](#)[Teams](#)[Help Center](#)[Blog](#)[Upgrade](#)

3. Crear un nuevo proyecto a partir de Github

Al dar click en “new project” nos desplegará una lista de plantillas de proyectos. Seleccionaremos el botón de abajo “Import from Github”.

Posteriormente, colocaremos el link que apunta a nuestro repositorio de Github.





APROXIMACIÓN AL PROCESO

Glitch.com comenzará a crear el proyecto y ajustarlo

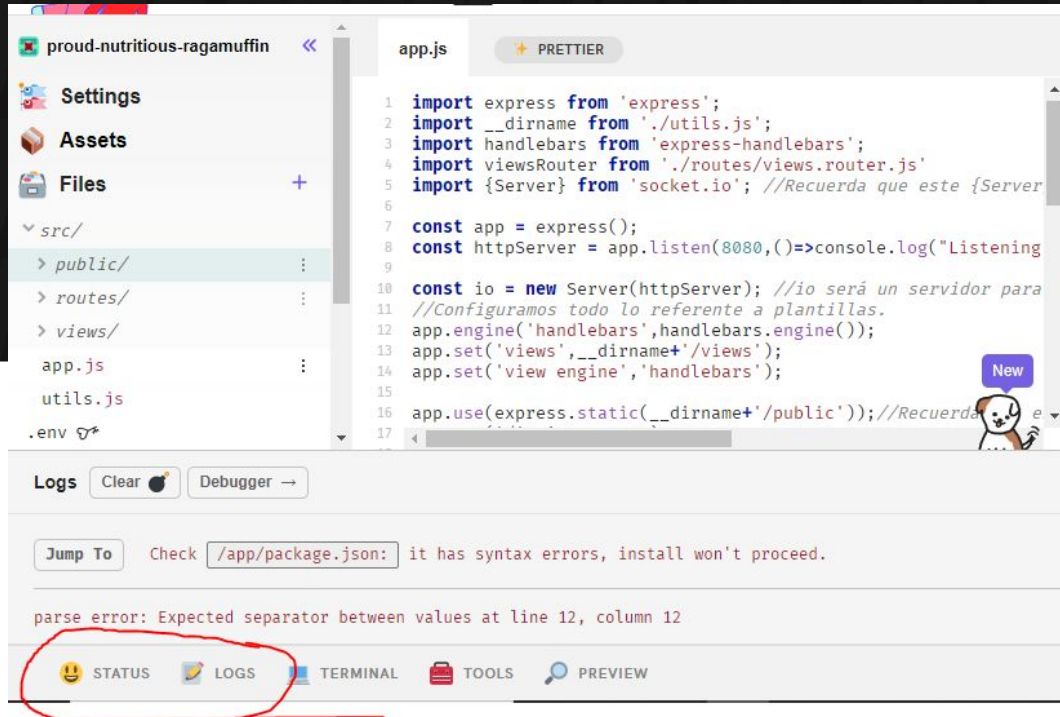
Loading Project



Glitch Tip!
Click "Show" to see your app

This site is protected by reCAPTCHA and the Google [Privacy Policy](#) and [Terms of Service](#) apply.

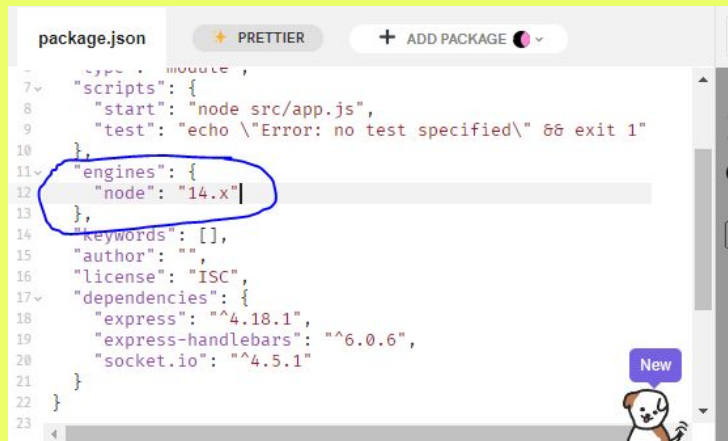
Revisa que el status esté bien, en caso contrario, revisar los LOGS





APROXIMACIÓN AL PROCESO

Tal vez hubo un error...



```
package.json
+ PRETTIER + ADD PACKAGE

1  "type": "module",
2  "scripts": {
3    "start": "node src/app.js",
4    "test": "echo \\\"Error: no test specified\\\" && exit 1"
5  },
6  "engines": {
7    "node": "14.x"
8  },
9  "keywords": [],
10 "author": "",
11 "license": "ISC",
12 "dependencies": {
13   "express": "^4.18.1",
14   "express-handlebars": "^6.0.6",
15   "socket.io": "^4.5.1"
16 }
17
18
19
20
21
22
23
```

Si los logs te marcan un error, es porque seguramente el core de nodejs no es correcto, prueba cambiando desde el package.json la versión del engine de nodejs a 14.x o 15.x



APROXIMACIÓN AL PROCESO

Finalmente, compartimos el link a quien queramos

Una vez que está andando el proyecto en Glitch.com, daremos click en el botón “Share”.



Este nos desplegará una lista de información, dentro de la cual los **project links** contienen lo que nos interesa: la opción **Live site** es el link que hará que cualquier persona a la que pases el link, pueda acceder al chat grupal.



¿Preguntas?

Muchas gracias.

#DemocratizandoLaEducación