



Clase 11. Programación Backend

Websockets



OBJETIVOS DE LA CLASE

- Comprender la diferencia entre HTTP y WebSocket.
- Integrar WebSocket a nuestro proyecto de Express.
- Generar la inicialización sobre el cliente para conectarse al servidor mediante WebSocket

CRONOGRAMA DEL CURSO

Clase 10



Pug & Ejs

Clase 11



Websockets

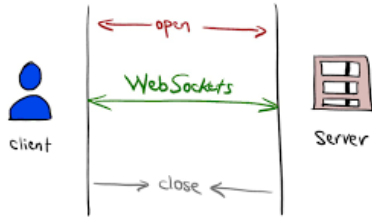
Clase 12



**Aplicación chat con
websocket**

Websocket





¿Qué es WebSocket?



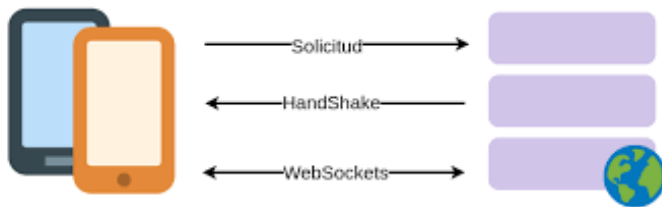
- WebSocket es un **protocolo de red basado en TCP** que establece cómo deben intercambiarse datos entre redes.
- Es un protocolo fiable y eficiente, utilizado por prácticamente todos los clientes.
- El protocolo TCP **establece conexiones entre dos puntos finales** de comunicación, **llamados sockets**.
- De esta manera, el **intercambio** de datos puede producirse en las **dos direcciones**.



¿Qué es WebSocket?

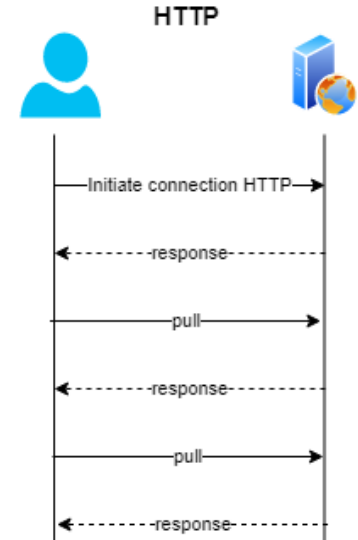
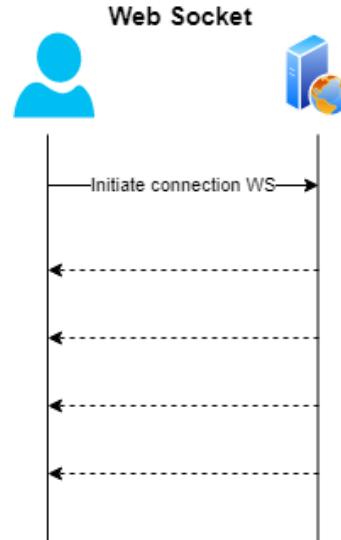
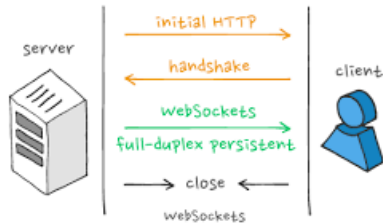
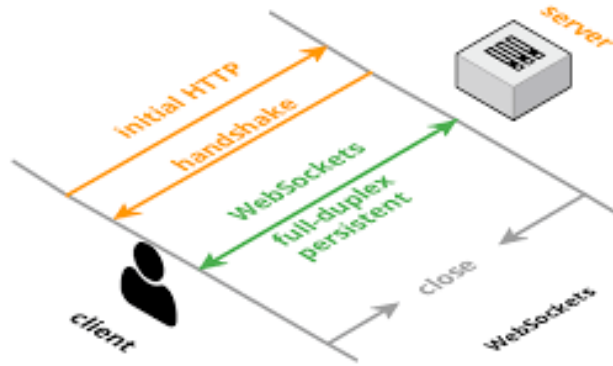


- En las **conexiones bidireccionales**, como las que crea WebSocket, se intercambian datos en ambas direcciones al mismo tiempo.
- La ventaja de usar WebSocket es **acceder de forma más rápida a los datos**.
- WebSocket permite una **comunicación directa y en tiempo real** entre una aplicación web y un servidor WebSocket.





Esquemas

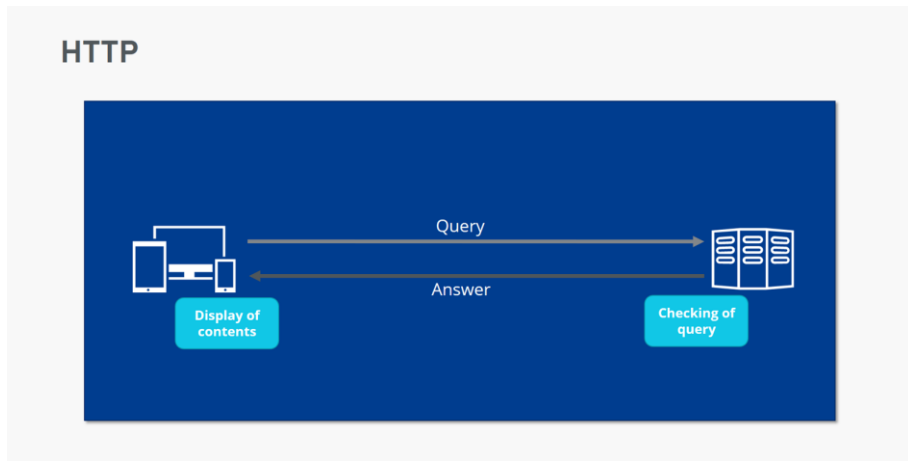
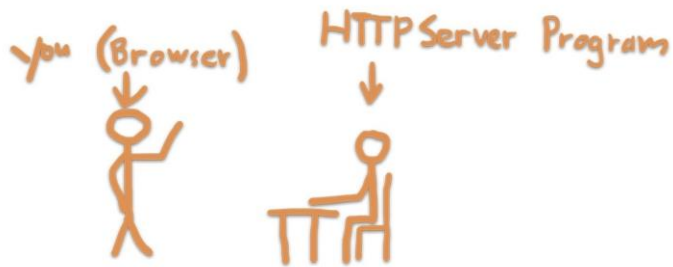
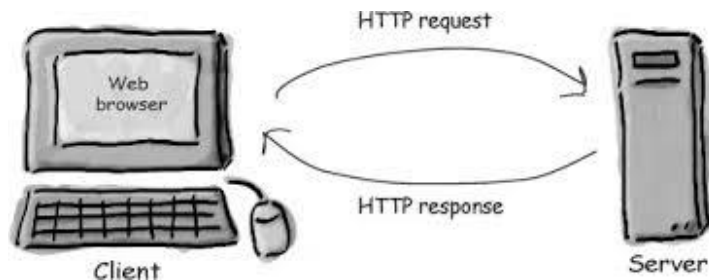


¿Cómo funciona un Websocket?

¿Cómo se accede a una página web sin WebSocket?

- La transmisión de páginas web en Internet suele realizarse mediante una **conexión HTTP**. Este protocolo sirve para transmitir datos y hace posible la carga de las páginas web en el navegador. Para lograrlo el **cliente** envía, con cada acción del usuario, una **solicitud** al **servidor**.
- Una vez enviada la solicitud, el servidor puede responder y mostrar el contenido solicitado. Se trata de un **rígido patrón de solicitud y respuesta** que provoca **largos tiempos de espera**.

Conexiones HTTP



Las conexiones **HTTP** se basan en el clásico esquema de pregunta y respuesta, en el que el cliente debe enviar una solicitud al servidor para que este pueda mostrar el contenido solicitado.

El protocolo WebSocket: principios



- WebSocket permitió por primera vez **acceder** a una **web** de **forma dinámica** en **tiempo real**.
- Basta con que el **cliente establezca** una **conexión** con el **servidor**, que se confirma mediante el llamado *apretón de manos* o WebSocket Protocol Handshake.
- Con él, el **cliente envía** al **servidor** todos los **datos de identificación** necesarios para el intercambio de información.

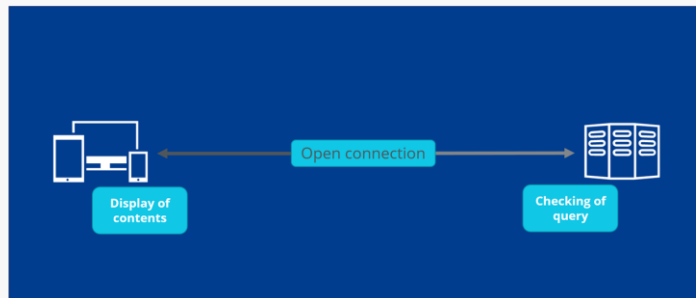
El protocolo WebSocket: principios



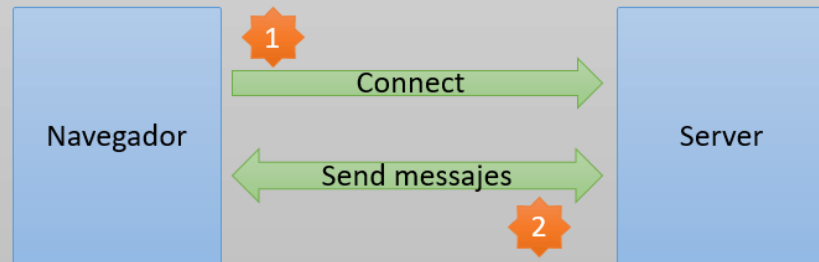
- El **canal** de comunicación queda “**abierto**” tras el handshake.
- El **servidor** puede **activarse por sí mismo** y poner toda la información a disposición del cliente, sin que este tenga que pedírselo. Si dispone de nueva información, se lo comunica al cliente, **sin necesidad de recibir una solicitud** específica para ello.
- Las **notificaciones push** de las páginas web funcionan según este principio.

Conexiones WebSocket

WebSocket



WebSocket interaction



WebSocket puede entenderse como un **canal de comunicación abierto**, en el cual queda abierta una **conexión activa tras el handshake** inicial entre el cliente y el servidor. Así, el servidor también puede enviar información nueva al cliente sin que este tenga que solicitarlo previamente cada vez.

Detalle de intercambio de datos

Para iniciar el intercambio con Websocket el **cliente envía una solicitud**, al igual que en el clásico HTTP. Sin embargo, la **conexión se establece mediante TCP y permanece abierta** tras el handshake entre el cliente y el servidor.

```
1 GET /chatService HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6 Sec-WebSocket-Origin: http://example.com
7 Sec-WebSocket-Protocol: chat, superchat
8 Sec-WebSocket-Version: 13
```

Detalle de intercambio de datos

El servidor responde:

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: WebSocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5 Sec-WebSocket-Protocol: superchat
```

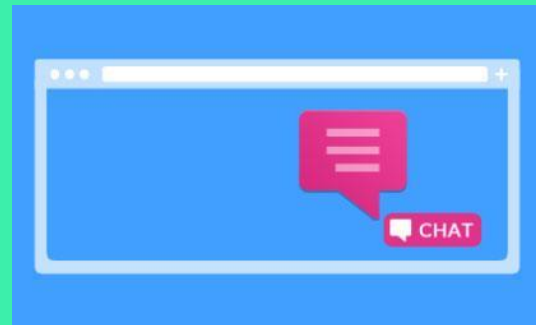
El nuevo esquema URL de Websocket para las páginas web mostradas se define con el **prefijo ws** en lugar de http. El prefijo que corresponde a una conexión segura es **wss**, de forma análoga a https.

¿Para qué se utiliza Websocket?



- Para establecer **conexiones de forma rápida**. Por ejemplo: chats de asistencia técnica, tickers de noticias o de actualizaciones de bolsa en directo, servicios de mensajería instantánea y juegos en tiempo real.
- Websocket también resulta muy útil en las **redes sociales** para establecer conexiones en directo con otras personas, así como para enviar y recibir mensajes instantáneos. Permite obtener altas velocidades de transmisión y limitar los tiempos de latencia.

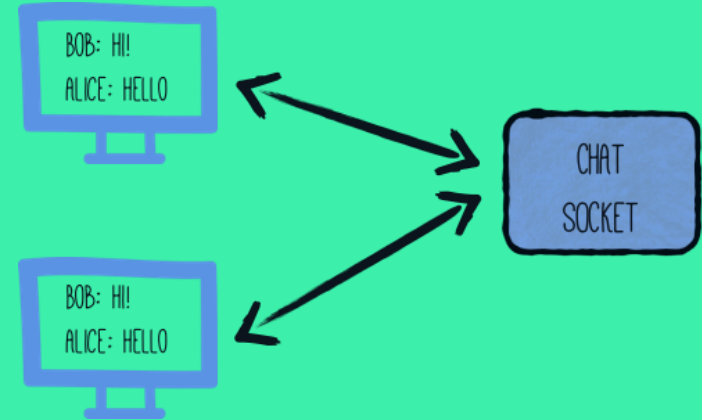
Desventajas uso de Http en chat



- El uso tradicional de las conexiones HTTP tiene el inconveniente de que el **cliente** siempre **carga la página HTML entera**.
- Para resolver el problema se desarrolló la **tecnología AJAX**. No obstante, trae la desventaja de establecer **conexiones unidireccionales**. Al permitir la comunicación en una sola dirección daría lugar a **largos tiempos de espera** en las intensivas aplicaciones de hoy en día, especialmente en los chats.

Ventajas uso de Websocket en chat

- Websocket crea **conexiones bidireccionales** que permiten el intercambio de datos en ambos sentidos, lo cual hace posible el contacto directo con el navegador y, con ello, permite **cortos periodos de carga**.
- En cuanto se envía un mensaje, como podría ser uno en un chat de soporte técnico, este llega y se muestra directamente al otro lado.



Websocket: comparación con HTTP



Fuente: EdTeam

WebSocket: ejemplos de usos útiles



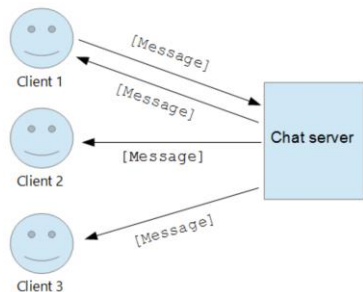
WebSocket se utiliza en **aplicaciones** que **requieren** una **conexión en tiempo real** entre el **cliente** y el **servidor** para poder así ofrecer sus servicios sin complicaciones.

Por ejemplo, algunos de los casos:

- Juegos en línea
- Plataformas de compra y de venta, como eBay
- Chats de atención al usuario
- Tickers de noticias deportivas en directo
- Actualizaciones en tiempo real de las redes sociales

Resumen

- Websocket **no es** un **sustituto total** de **HTTP**, pero puede usarse como **canal** de comunicación **eficiente y bidireccional** siempre que se necesite dar o recibir **información** en **tiempo real**.
- El protocolo **Websocket** está muy vinculado con el desarrollo de **HTML5**: un intento de hacer la **web más rápida**, más **dinámica** y más **segura**.
Permite a las aplicaciones web reaccionar mucho más rápido que con la comunicación HTTP convencional. Sin embargo, esto no significa que haya que reemplazar el protocolo tradicional: a pesar de la existencia de Websocket, **HTTP sigue siendo un estándar clave** en Internet.

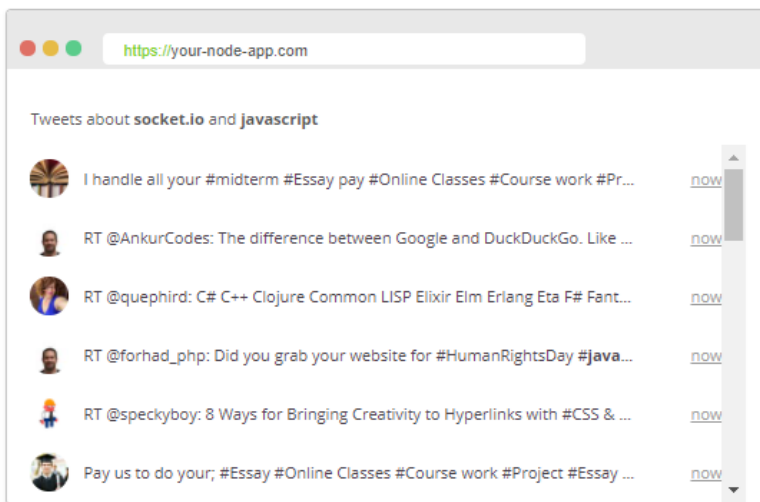


SOCKET.IO



```
~/Projects/tweets/index.js

1. const io = require('socket.io')(80);
2. const cfg = require('./config.json');
3. const tw = require('node-tweet-stream')(cfg);
4.
5. tw.track('socket.io');
6. tw.track('javascript');
7.
8. tw.on('tweet', (tweet) => {
9.   io.emit('tweet', tweet);
10. });
```



CODER HOUSE

¿Que es Socket.io?



socket.io



Socket.IO es una **biblioteca de JavaScript** para aplicaciones web en tiempo real. Permite la comunicación bidireccional en tiempo real entre servidores y clientes web.



¿Que es Socket.io?



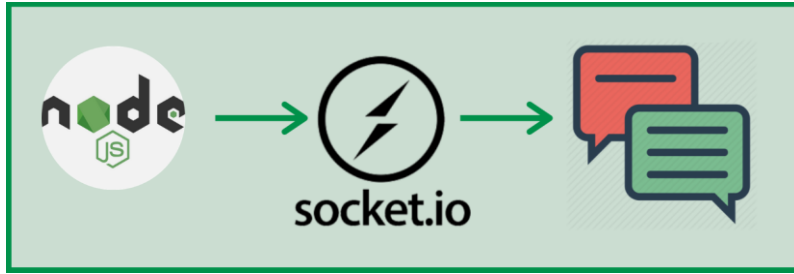
socket.io



Tiene dos partes:

- Una biblioteca del lado del cliente que se ejecuta en el navegador.
- Una biblioteca del lado del servidor para Node.js.

Ambos componentes tienen una API casi idéntica. Al igual que Node.js, está impulsado por eventos.



Socket.io: características



- Socket.IO **utiliza principalmente** el protocolo **Websocket** proporcionando la misma interfaz.
- Se puede **usar como** un **contenedor** para Websocket aunque proporciona muchas más funciones, incluida la transmisión a múltiples sockets, el almacenamiento de datos asociados con cada cliente y E/S asíncronas.
- Se puede instalar con **npm**.



Socket.io: Características



Sus principales características son:

- **Fiabilidad:** Las conexiones se establecen incluso en presencia de:
 - proxies y balanceadores de carga.
 - firewall personal y software antivirus.
- **Soporte de reconexión automática:** A menos que se le indique lo contrario, un cliente desconectado intentará siempre volver a conectarse, hasta que el servidor vuelva a estar disponible.



Socket.io: Características



- **Detección de desconexión:** Se implementa un mecanismo de heartbeat, lo que permite que tanto el servidor como el cliente sepan cuando el otro ya no responde.
- **Soporte binario:** Se puede emitir cualquier estructura de datos serializable, que incluye:
 - ArrayBuffer y Blob en el navegador
 - ArrayBuffer y Buffer en Node.js



BREAK

¡5/10 MINUTOS Y VOLVEMOS!

Implementación Socket.io en Node.js



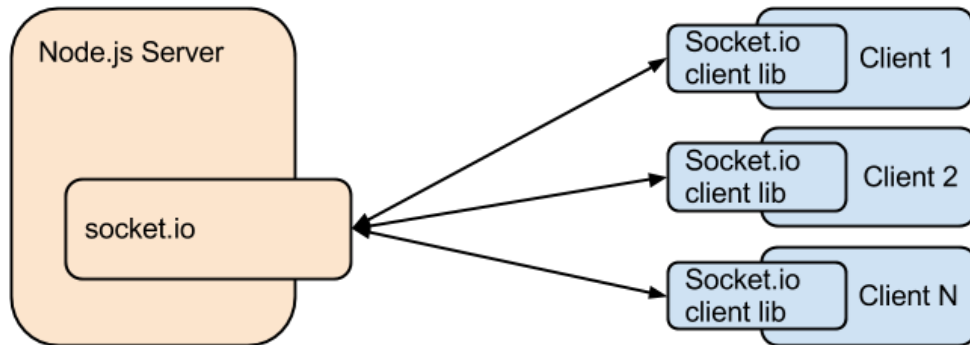
Introducción



Socket.IO permite la **comunicación bidireccional basada en eventos en tiempo real**.

Consiste en

- Un servidor Node.js
- Una librería cliente de Javascript para el navegador o un cliente Node.js





Pasos a seguir

1. Crear nuestro directorio de trabajo e instalar socket.io y express.
2. Una vez instalados dichos módulos agregamos un archivo index.html y un server.js
3. Crearemos una carpeta (en este caso se llamará public). En esta carpeta agregaremos un index.js y un archivo style.css



Creación de proyecto e instalación de módulo

Vamos a realizar la **inicialización** del proyecto e **instalación** de los módulos.

```
npm init -y && npm install express  
socket.io
```

```
.  
|- node_modules  
|- public  
|   |-index.js  
|   |-style.css  
|-index.html  
|-server.js  
|-package-lock.json  
|-package.json
```

Luego **configuramos** nuestro archivo **server.js** importando en primer lugar los módulos que instalamos anteriormente.

```
const express = require('express')  
const { Server: IOServer } = require('socket.io')
```




Inicializaciones de proyecto

Vamos a inicializar la función `express()`. También importaremos el módulo `http` que será necesario para que nuestros sockets funcionen.

Complementando la parte de las importaciones e inicializaciones el código quedaría de la siguiente forma.

```
const express = require('express')
const { Server: HttpServer } = require('http')
const { Server: IOserver } = require('socket.io')

const app = express()
const httpServer = new HttpServer(app)
const io = new IOserver(httpServer)
```



Inicialización del servidor

Configuramos el middleware para dejar disponibles las rutas al igual que los archivos estáticos, así el archivo index.html se mostrará al ingresar a la página. **Arrancamos el servidor con `http.listen()` y NO con `app.listen()`**

```
//...  
// Indicamos que queremos cargar los archivos estáticos que se encuentran en dicha  
carpeta  
app.use(express.static('./public'))  
// Esta ruta carga nuestro archivo index.html en la raíz de la misma  
app.get('/', (req, res) => {  
  res.sendFile('index.html', { root: __dirname })  
})  
// El servidor funcionando en el puerto 3000  
httpServer.listen(3000, () => console.log('SERVER ON'))
```



Inicialización de la vista

En nuestro archivo index.html agregamos la estructura inicial de nuestro y también referenciamos al index.js y al style.css

Se incluye un script que contiene la configuración de los sockets y es importante para que funcionen en nuestro cliente. Dicho script es parte del módulo socket.io

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Sockets</title>
  <link rel="stylesheet" href="/estilos.css">
  <script src="/socket.io/socket.io.js"></script>
  <script src="/index.js"></script>
</head>
<body>
  <h1>TUTORIAL DE SOCKETS.IO</h1>
</body>
</html>
```



Inicialización del canal de Websocket

Cliente: A continuación procedemos a inicializar una constante en nuestro index.js que será el siguiente código

```
const socket = io(); // Ya podemos empezar a usar los sockets desde el cliente :)
```

Server: Una vez realizado lo anterior, nos vamos a nuestro archivo server.js y agregamos el siguiente código:

```
//...  
io.on('connection', (socket) => {  
  // "connection" se ejecuta la primera vez que se abre una nueva conexión  
  console.log('Usuario conectado')  
  // Se imprimirá solo la primera vez que se ha abierto la conexión  
})
```



Inicialización del server

El código del archivo server.js quedaría de la siguiente forma

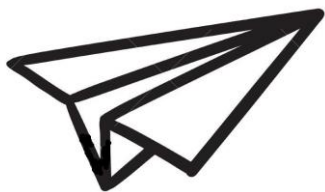
```
const express from 'express';
const { Server: HttpServer } from 'http'
const { Server: IOserver } from 'socket.io'

const app = express()
const httpServer = new HttpServer(app)
const io = new IOserver(httpServer)

app.use(express.static('./public'))
// Indicamos que queremos cargar los archivos estáticos que se encuentran en dicha carpeta
app.get('/', (req, res) => { // Esta ruta carga nuestro archivo index.html en la raíz de la misma
  res.sendFile('index.html', {root: __dirname})
})

httpServer.listen(3000, () => console.log('SERVER ON')) // El servidor funcionando en el puerto 3000

io.on('connection', (socket) => { // "connection" se ejecuta la primera vez que se abre una nueva conexión
  console.log('Usuario conectado') // Se imprimirá solo la primera vez que se ha abierto
})
```

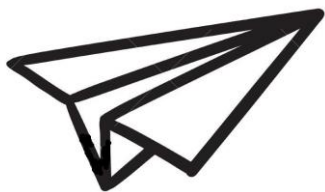


Envío de datos al cliente (servidor)

Vamos a **emitir un mensaje** de nuestro servidor al cliente

```
// Servidor
io.on('connection', socket => {
  console.log('Usuario conectado')
  socket.emit('mi mensaje', 'Este es mi mensaje desde el servidor')
})
```

Podemos ver que hemos utilizado el objeto socket y este a su vez contiene diversos métodos, entre ellos **emit()**. Este nos permite **enviar un mensaje del servidor al cliente**. El primer parámetro que recibe es el nombre de nuestro evento y el segundo parámetro es la información que queremos transmitir.



Recepción de datos del servidor (cliente)

Ahora continuando con nuestro evento personalizado. Veamos cómo podemos recibir la información en el cliente. En nuestro archivo .js agregamos el siguiente código.

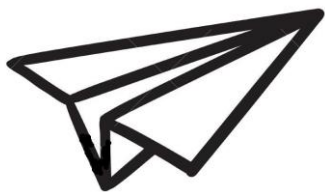
```
// Cliente
socket.on('mi mensaje', data => {
  alert(data)
})
```

localhost:3000 dice

Este es mi mensaje desde el servidor

Aceptar

Vemos que nos imprime un alert con el mensaje que recibimos desde el servidor.

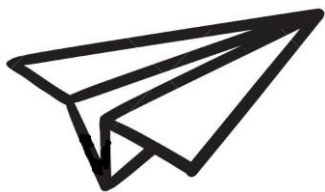


Envío de datos al servidor (cliente)

También podemos **enviar información del cliente al servidor** de forma bastante similar. Por ejemplo, supongamos que después de imprimir el alert queremos enviar un mensaje al servidor que notifique que el mensaje fue recibido con éxito.

```
// Cliente
socket.on('mi mensaje', data => {
  alert(data)
  socket.emit('notificacion', 'Mensaje recibido exitosamente')
})
```

Podemos ver que después del alert estamos emitiendo de la misma forma que lo hacíamos en el servidor.



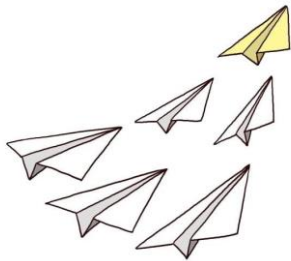
Recepción de datos del cliente (servidor)

Para recibirlo en el cliente sería de la siguiente forma:

```
// Servidor
socket.on('notificacion', data => {
  console.log(data)
})
```

En la consola del servidor se mostrará el mensaje recibido

```
Mensaje recibido exitosamente
```



Envío de datos a todos los clientes conectados (servidor)

```
io.on('connection', socket => {  
  console.log('¡Nuevo cliente conectado!');  
  
  /* Envío los mensajes al cliente que se conectó */  
  socket.emit('mensajes', mensajes);  
  
  /* Escucho los mensajes enviados por el cliente y se los propago a todos */  
  socket.on('mensaje', data => {  
    mensajes.push({ socketid: socket.id, mensaje: data });  
    io.sockets.emit('mensajes', mensajes);  
  });  
});
```

Utilizando el método **io.sockets.emit** enviamos un mensaje global a todos los clientes conectados al canal de Websocket



SERVIDOR CON WEBSOCKET

Vamos a practicar lo aprendido hasta ahora

SERVIDOR CON WEBSOCKET - 1

Desafío
generico



- 1) Desarrollar un servidor basado en express que tenga integrado Websocket. Con cada conexión de cliente, el servidor debe emitir por consola en mensaje: '¡Nuevo cliente conectado!'

Tiempo: 5 minutos

CODER HOUSE

SERVIDOR CON WEBSOCKET - 2

Desafío
generico



- 2) 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.

SERVIDOR CON WEBSOCKET - 3

Desafío
generico



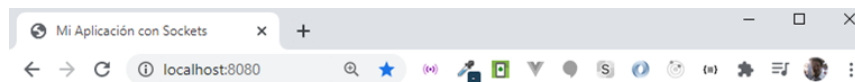
- 3) Basado en el ejercicio que venimos realizando, ahora los mensajes enviados por los clientes deberán ser almacenados en el servidor y reflejados por debajo del elemento de entrada de texto cada vez que el usuario haga un envío. La estructura de almacenamiento será un array de objetos, donde cada 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.

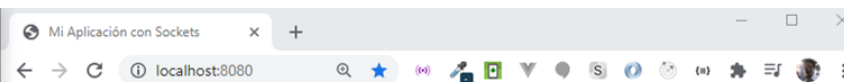
Tiempo: 15 minutos

CODER HOUSE



Cliente Websockets

SocketId: gSOwhnuxTvC1Nsf6AAAF -> Mensaje: hola
SocketId: CaaecnmWUlh_8Vq5AAAH -> Mensaje: Que tal
SocketId: gSOwhnuxTvC1Nsf6AAAF -> Mensaje: bien



Cliente Websockets

SocketId: gSOwhnuxTvC1Nsf6AAAF -> Mensaje: hola
SocketId: CaaecnmWUlh_8Vq5AAAH -> Mensaje: Que tal
SocketId: gSOwhnuxTvC1Nsf6AAAF -> Mensaje: bien

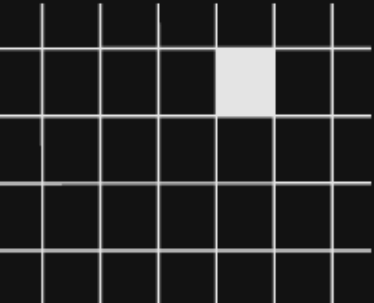
¿PREGUNTAS?





¡MUCHAS GRACIAS!

Resumen de lo visto en clase hoy:

- WebSocket
 - Socket.IO
- 



OPINA Y VALORA ESTA CLASE

#DEMOCRATIZANDO LA EDUCACIÓN