



Desarrollo de Aplicaciones Informáticas

Tema: Explicación y uso del hook `useSocket` en React

Docente: Ing. Pablo Morandi

Ayudante: Matías Marchesi

Curso: 5to año

Especialidad: Informática

Redactado por: Matías Marchesi

- 1. Preparativos
- 2. Introducción
- 3. Configuración del Backend
 - 3.1 Configuración inicial del servidor
 - 3.2 Implementación de eventos Socket.IO
- 4. Explicación del Frontend
- 5. ¿Cómo usarlo?
 - 5.1 Evento PingAll
 - 5.2 Evento joinRoom
 - 5.3 Evento sendMessage
 - 5.4 Evento personalizado
- 6. Como levantar múltiples puertos en el front
 - 6.1 Prueba con múltiples puertos

1. Preparativos

1. Descargar el hook `useSocket.js` del classroom.
2. Crear la carpeta `hooks` en el frontend dentro de `src/` y colocar allí el archivo `useSocket.js`.
3. Descargar el backend proporcionado en el classroom.
4. Configurar en `useSocket.js` la **IP del backend** correspondiente.
5. Instalar dependencias necesarias:

```
# Backend (ya instalada en el back dado)
npm i socket.io
```

```
# Frontend (NextJS)
npm i socket.io-client
```

2. Introducción

WebSocket permite establecer una **comunicación bidireccional** en tiempo real entre cliente y servidor, a diferencia de las peticiones HTTP tradicionales (**GET** , **POST** , **PUT** , **DELETE**).

Esto nos habilita a intercambiar datos instantáneamente, mantener sincronización entre varios clientes y desarrollar aplicaciones que requieran **actualización en tiempo real**.

3. Configuración del Backend

3.1 Configuración inicial del servidor

El backend requiere una configuración específica para poder manejar conexiones WebSocket junto con Express y las sesiones:

```
const port = process.env.PORT || 4000; // Puerto por el que estoy ejecutando la página Web

const cors = require("cors");
const session = require("express-session"); // Para el manejo de las variables de sesión

app.use(cors());

const server = app.listen(port, () => {
  console.log(`Servidor NodeJS corriendo en http://localhost:${port}/`);
});

const io = require("socket.io")(server, {
  cors: {
    // IMPORTANTE: REVISAR PUERTO DEL FRONTEND
    origin: ["http://localhost:3000", "http://localhost:3001"], // Permitir el origen localhost:3000
    methods: ["GET", "POST", "PUT", "DELETE"], // Métodos permitidos
    credentials: true, // Habilitar el envío de cookies
  },
});

const sessionMiddleware = session({
  //Elegir tu propia key secreta
  secret: "supersarasa",
  resave: false,
  saveUninitialized: false,
});

app.use(sessionMiddleware);

io.use((socket, next) => {
  sessionMiddleware(socket.request, {}, next);
});
```

Puntos importantes de esta configuración:

- **Puerto:** El servidor corre en el puerto 4000 por defecto
- **CORS:** Configurado para permitir conexiones desde los puertos 3000 y 3001 del frontend
- **Sesiones:** Integración entre Express sessions y Socket.IO para mantener estado de usuario
- **Middleware:** Permite acceder a las variables de sesión desde los eventos de Socket.IO

3.2 Implementación de eventos Socket.IO

Una vez configurado el servidor, implementamos los eventos que manejarán la comunicación en tiempo real:

```
/*
 A PARTIR DE ACÁ LOS EVENTOS DEL SOCKET
 A PARTIR DE ACÁ LOS EVENTOS DEL SOCKET
 A PARTIR DE ACÁ LOS EVENTOS DEL SOCKET
 */

io.on("connection", (socket) => {
    const req = socket.request;

    socket.on("joinRoom", (data) => {
        console.log("🚀 ~ io.on ~ req.session.room:", req.session.room);
        if (req.session.room != undefined && req.session.room.length > 0)
            socket.leave(req.session.room);
        req.session.room = data.room;
        socket.join(req.session.room);

        io.to(req.session.room).emit("chat-messages", {
            user: req.session.user,
            room: req.session.room,
        });
    });

    socket.on("pingAll", (data) => {
        console.log("PING ALL: ", data);
        io.emit("pingAll", { event: "Ping to all", message: data });
    });

    socket.on("sendMessage", (data) => {
        io.to(req.session.room).emit("newMessage", {
            room: req.session.room,
            message: data,
        });
    });

    socket.on("disconnect", () => {
        console.log("Disconnect");
    });
});
```

Explicación de cada evento:

- **connection**: Se ejecuta automáticamente cuando un cliente se conecta
 - **joinRoom**: Permite cambiar de sala, saliendo de la anterior y uniéndose a la nueva
 - **pingAll**: Envía un mensaje a todos los clientes conectados
 - **sendMessage**: Envía mensajes solo a los clientes de la misma sala
 - **disconnect**: Se ejecuta cuando un cliente se desconecta
-

4. Explicación del Frontend

En el backend tenemos definidos **eventos**, que son las acciones con las que el cliente se comunica con el servidor:

- **connection**: se establece automáticamente al importar el hook.
 - **pingAll**: envía un mensaje a todos los clientes conectados.
 - **joinRoom**: permite unirse a una sala (`room`).
 - **sendMessage**: envía mensajes a todos los clientes de la sala.
 - **disconnect**: se dispara automáticamente al finalizar la conexión.
-

5. ¿Cómo usarlo?

Ejemplo creando una página nueva en el directorio `socket` en el frontend:

```
import { useSocket } from "@/hooks/useSocket";
import { useEffect } from "react";

export default function SocketPage() {
  const { socket, isConnected } = useSocket();

  useEffect(() => {
    if (!socket) return;
    //Aquí entrará cuando reciba un evento
  }, [socket]);

  return <h1>Socket funcionando</h1>;
}
```

Importando el hook y declarando `socket` e `isConnected`, al conectarnos en la página debería aparecer **Web Socket Conectado** en la consola.

5.1 Evento PingAll

En el frontend definimos un botón que emita el evento:

```
function pingAll() {  
  socket.emit("pingAll", { msg: "Hola desde mi compu" });  
}
```

El backend recibe el evento `pingAll`, lo emite a todos los clientes conectados y en el frontend debemos **escuchar** el evento:

```
socket.on("pingAll", (data) => {  
  console.log(data);  
});
```

Ejemplo de mensaje recibido:

```
{ "event": "pingAll", "message": { "msg": "Hola desde mi compu" } }
```

5.2 Evento joinRoom

Permite unirse a una sala determinada:

```
socket.emit("joinRoom", { room: "pio" });
```

El backend:

- Sale de la sala anterior.
 - Une al cliente a la nueva sala.
 - Emite un evento `chat-messages` notificando que un usuario ingresó.
-

5.3 Evento sendMessage

Creamos un `input` y un botón para enviar mensajes:

```
const [message, setMessage] = useState("");  
  
function sendMessage() {  
  socket.emit("sendMessage", { message });  
}
```

El backend recibe `sendMessage` y emite:

```
{  
  "room": "pio",  
  "message": { "message": "Mensaje a la sala" }  
}
```

El frontend escucha con:

```
socket.on("newMessage", (data) => {  
  console.log(data);  
});
```

5.4 Evento personalizado

Podemos crear eventos propios. Ejemplo:

Backend

```
let contador = 0;  
  
socket.on("eventoPersonalizado", () => {  
  contador++;  
  socket.emit("respuestaPersonalizada", { contador });  
});
```

Frontend

```
function emitirEvento() {  
  socket.emit("eventoPersonalizado");  
}  
  
useEffect(() => {  
  socket.on("respuestaPersonalizada", (data) => {  
    setContador(data.contador);  
  });  
}, [socket]);
```

6. Como levantar múltiples puertos en el front

Abrir otra terminal y ejecutar:

```
npm run dev
```

Si el puerto **3000** ya está en uso, Next levantará la app en **3001**.

El backend está configurado para aceptar conexiones desde **3000 y 3001**.

6.1 Prueba con múltiples puertos

Abrir dos clientes:

- Uno en **http://localhost:3000**
- Otro en **http://localhost:3001**

Probar:

- **pingAll** : se emitirá a ambos clientes.
 - **sendMessage** : solo los clientes que estén en la misma sala (**pio**) recibirán el mensaje.
-

Con esto finaliza la explicación y uso de **useSocket en React**.