El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

CICLO IV:

Desarrollo de Aplicaciones Web

MisionTIC2022

El futuro digital es de todos | MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Sesión 12:

Desarrollo de Aplicaciones Web

Desarrollo de Back-End web con Node.js

MisiónTIC2022

Objetivos de la sesión

Al finalizar esta sesión estarás en capacidad de:

1. Crear una aplicación de prueba con Express.js.
2. Implementar websockets en una aplicación web con Node.js.

Node.js - Express.js

- Por lo general uno puede construir un servidor web sin necesidad de librerías utilizando únicamente las utilidades de Node.js.
- Esto se puede lograr mediante el paquete `http` de Node.js.
- Como podemos ver en el siguiente ejemplo. *Adaptado de este enlace*:

```
const http = require('http');
http.createServer((request, response) => {
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write('Hola mundo!');
    response.end();
}).listen(3000);
```

- Se puede crear un servidor web en un puerto deseado, y para enviar una respuesta tenemos que manualmente escribir los encabezados el cuerpo y cerrar el buffer de escritura.

Node.js - Express.js

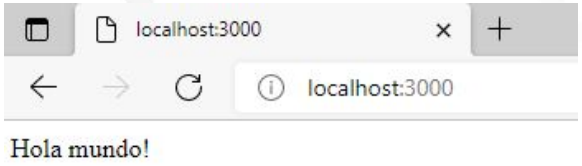
- Esto puede llegar a ser engorroso ya que nos tocaría configurar cada solicitud de forma manual.
- Para esto existen librerías como express que en últimas trabaja por encima de un servidor creado con http, pero añadiendo más funcionalidades como alojamiento de archivos estáticos.
- A modo de comparación haremos el mismo ejemplo anterior pero en express:

```
const express = require('express')
const app = express()
app.get('/', (req, res) => { res.send('Hola mundo!'); })
app.listen(3000)
```

- Como podemos ver esto simplifica la lógica de lado del servidor.

Node.js - Express.js

- Ahora si analizamos detalladamente el ejemplo anterior notaremos que con el uso de app.get definimos una ruta ‘/’, donde nuestro servidor aceptará solicitudes `http` con el método GET.
- Asi mismo tambien usamos un callback para definir el funcionamiento de nuestro recurso.
- Por último podremos revisar nuestro recurso en **`localhost:3000/`**.



Node.js - Websockets

- Es posible que un servidor `REST` no sea suficiente para los requerimientos de nuestra app.
- Ya que si quisiéramos tener una aplicación que funcione con eventos en tiempo real, como un chat, no podríamos trabajar con recursos `HTTP`.
- Esto se debe a que de esta forma el servidor no podría comunicarse con el cliente sin haber recibido una solicitud `HTTP` del cliente primero.
- Para esto se introducen los WebSockets.
- Entre las implementaciones más populares de sockets encontramos a:
  - Socket.io.
  - ws.
  - uWebSocket.

Node.js - Websockets

- Esto nos permite modificar la forma en la que nuestro servidor interactúa con los clientes:

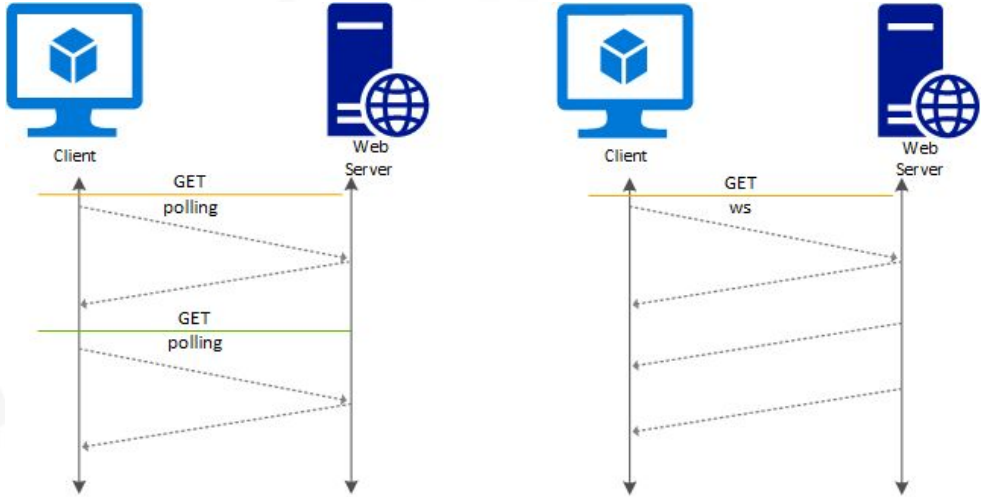


Imagen tomada de Medium/@xanderbakx

Node.js - Chat App Servidor

- Asumamos que queremos construir un servidor web para un chat con Express y Socket.io.
- Para lo siguiente crearemos un nuevo proyecto con node mediante **`npm init -y`**.
- Luego instalamos las siguientes dependencias:
  - Express, nuestro servidor web.
  - Compression, un middleware o intermediario para express.
  - Socket.io, nuestro servidor de sockets.
- Con el comando **`npm i --save express compression socket.io`**.

Node.js - Chat App Servidor

- Así mismo agregaremos las siguientes dependencias de desarrollador:
  - Babel, para compilar nuestro javascript de una versión moderna a una más antigua.
  - Nodemon, es un proceso que nos permitirá reiniciar el servidor automáticamente al hacer cambios.

- Con el comando **`npm i --save-dev @babel/cli @babel/core @babel/node @babel/preset-env nodemon.`**

- Luego procedemos a crear nuestro archivo de configuración para babel, **`.babelrc`**, de la siguiente forma:

```
{
    "presets": [["@babel/preset-env", { "targets": { "node": true } }]]
}
```

Node.js - Chat App Servidor

- Luego modificaremos nuestro package.json donde añadiremos los siguientes comandos:

```
"start": "npm run build && node ./build/server.js",
"start:dev": "nodemon --exec babel-node src/server.js",
"build-server": "babel -d ./build ./src -s",
"build": "npm run build-server"
```

- Los comandos funcionan de la siguiente forma:
  - **`start`**: Genera el compilado de nuestro servidor y lo ejecuta con node.
  - **`start:dev`**: Ejecuta un servidor con babel que observa los cambios y se reinicia.
  - **`build-server`**: Se encarga de compilar nuestro proyecto a la carpeta build
  - **`build`**: Se encarga de ejecutar el comando build-server
- Luego procedemos a crear nuestro archivo de inicio **`src/server.js`**.

Node.js - Chat App Servidor

- src/server.js:

```
import express from 'express';
import compression from 'compression';
import { createServer } from 'http';
import { configureSocketSever, reset, users } from './socket-server';

const PORT = process.env.PORT || 3000;
const app = express();
const httpServer = createServer(app);
configureSocketSever(httpServer);
app.use(compression());

app.use('/client', express.static('public'));
app.get('/status', (req, res) => { res.json(users); });
app.get('/reset', (req, res) => { reset(); res.json({ message: 'ok' }); });
httpServer.listen(PORT, () =>
    console.log(`Sevidor esperando por peticiones en localhost:${PORT}`)
);
```

Node.js - Chat App Servidor

- Creamos un servidor con **`express`**, al cual le agregamos el middleware de **`compression`**, y **`http`**.
- Adicionalmente importamos unos métodos de nuestro archivo **`src/socket-server.js`**.
- Definimos nuestra ruta **`/client`** donde alojaremos nuestro cliente web el cual se encuentra en la carpeta **`/public`**.

Node.js - Chat App Servidor

- Definimos nuestra ruta **`/status`** donde recibiremos un json de los usuarios que se encuentran activos o han usado el chat recientemente.
- Definimos nuestra ruta **`/reset`** donde iniciaremos el listado de usuarios.
- Finalmente agregamos un console.log para indicar que el servidor se encuentra activo.

Node.js - Chat App Servidor

- En nuestro archivo src/socket-server.js:
  - Importamos la clase `Server` de `Socket.io`:

```
import { Server } from 'socket.io';
```

  - Definimos nuestro listado de usuario y sockets, así mismo como un método para reiniciarlos:

```
let users = {};
let sockets = {};

export const reset = () => {
    for (const user of Object.keys(sockets)) {
        sockets[user].disconnect();
    }
    users = {};
    sockets = {};
};
```

Node.js - Chat App Servidor

- En nuestro archivo **src/socket-server.js**:
  - Creamos nuestro socket server:

```
export const configureSocketSever = httpServer => {
    const io = new Server(httpServer);

    io.on('connection', socket => {
        /**
         * Aquí va la lógica de nuestro socket
         */
    });
};
```

  - Luego definimos nuestra lógica de conexión:

Node.js - Chat App Servidor

- En nuestro archivo `src/socket-server.js`:
  - Luego definimos nuestra lógica de conexión:

```
const { user } = socket.handshake.query;
const session = { id: socket.id, user: user };
const userExists = Boolean(users[session.user]);
if (userExists && users[session.user] !== session.id) {
    console.log(`[INFO] El ${session.user} ya tiene una sesión! Cerrando
conexión`);
    socket.disconnect();
} else {
    console.log(`[INFO] Se ha conectado el usuario ${session.user}!`);
    users[session.user] = session.id;
    sockets[session.id] = socket;
    socket.broadcast.emit('user-connect', { user: session.user });
}
```

- Aceptaremos a un usuario y emitiremos el evento “user-connect” siempre y cuando no tengamos una conexión activa con dicho usuario.

Node.js - Chat App Servidor

- En nuestro archivo `src/socket-server.js`:
  - Agregamos “ping” a nuestro socket, donde emitiremos “pong”:

```
socket.on('ping', () => { socket.emit('pong'); });
```

  - Agregamos “disconnect” a nuestro socket:

```
socket.on('disconnect', () => {
        console.log(`[INFO] El usuario ${session.user} se ha desconectado.`);
});
```

Node.js - Chat App Servidor

- En nuestro archivo `src/socket-server.js`:
  - Agregamos “close” a nuestro socket, donde emitiremos “user-disconnect”:

```
socket.on('close', () => {
    users[session.id] = null;
    users[session.user] = null;
    socket.broadcast.emit('user-disconnect', { user: session.user });
    socket.disconnect();
});
```

  - Agregamos “client-message” a nuestro socket, donde emitiremos “server-message”:

```
socket.on('client-message', ({ message }) => {
    console.log(`[INFO] ${session.user} enviado un mensaje!`);
    socket.broadcast.emit('server-message', {
        user: session.user,
        message: message,
    });
});
```

Node.js - Chat App Servidor

- Como podemos ver se sigue manteniendo una conexión cliente servidor donde el cliente envía mensajes al servidor.
- Y el servidor le envía mensajes al cliente como lo es en el caso de los eventos “user-connect”, “pong”, “user-disconnect”, “server-message”.
- Esto ocurre sin que el cliente o el servidor se quede esperando por una respuesta del otro.
- Y así puede llevar a cabo otras tareas y procesar los comandos que le lleguen a través socket en tiempo real.

Node.js - Chat App Cliente

- Nuestro servidor se encargará de alojar nuestro cliente mediante archivos estáticos que se encontraran en la carpeta **`public/`**:

```
|-- public/
| |-- css/
| | `-- styles.css
| |-- js/
| | |-- client.js
| | `-- script.js
| `-- index.html
```

- Donde en `index.html` solo definiremos la interfaz e importaremos nuestros estilos, estilos de bootstrap y de animate.css.
- En nuestro archivo **`script.js`** y **`client.js`** es donde reside toda la lógica.

Node.js - Chat App Cliente

- En **public/js/script.js**
  - Definiremos una clase Client donde en agregaremos el siguiente constructor:

```
constructor() {
    let btn = document.getElementById('connect');
    let userNameInput = document.getElementById('user');

    btn.onclick = () => { this.startChat(userNameInput.value); };

    userNameInput.addEventListener('keypress', e => {
        let key = e.key || e.which || e.keyCode;
        if (key === 'Enter' || key === 13) {
            this.startChat(userNameInput.value);
        }
    });
}
```

- Donde ejecutaremos el método startChat.

Node.js - Chat App Cliente

- En `public/js/script.js`
  - Definiremos una clase Client con el método startChat:

```
startChat(user) {
    const isValid = this.isInputValid(user);
    if (!isValid) return;

    let btn = document.getElementById('connect');
    btn.disabled = true;
    const client = new SocketClient(user);
    client.initialize();
}
```

- Donde ejecutaremos el método **`isInputValid`** solo para validar nuestro campo de usuario. Cabe resaltar que solo se valida que su longitud sea mayor a 3, sino retornamos false.

Node.js - Chat App Cliente

- En p**ublic/js/script.js**
  - Por último cuando nuestro documento termine de cargar ejecutaremos lo siguiente:

```
window.onload = () => {
    new Client();
};
```

- Donde se creará un nuevo Client y dará inicio a todo el flujo de los sockets.

Node.js - Chat App Cliente

- En **public/js/client.js**
  - Aquí sólo nos detendremos a analizar los métodos initialize, setupEvents, emit, close y sendMessage.
  - En nuestro método initialize creamos un socket de la siguiente forma:

```
this.socket = io({
    query: { user: this.user },
});
```

  - Añadimos “connect_failed” al socket:

```
this.socket.on('connect_failed', () => {
    this.socket.close();
    console.log('[INFO] No se pudo establecer conexión!');
});
```

  - Añadimos “disconnect” al socket, donde solo actualizaremos la interfaz gráfica.

Node.js - Chat App Cliente

- En `public/js/client.js`
  - Añadimos “pong” a nuestro socket:

```
this.socket.on('pong', () => {
    console.log('pong!');
    this.addSystemMessage(`<span>pong!</span>`);
});
```

  - Añadimos “user-connect” al socket:

```
this.socket.on('user-connect', ({ user }) => {
    this.addSystemMessage(`<span>${user}</span> se ha unido`);
});
```

  - Añadimos “user-disconnect” al socket:

```
this.socket.on('user-disconnect', ({ user }) => {
    this.addSystemMessage(`<span>${user}</span> se ha desconectado`);
});
```

Node.js - Chat App Cliente

- En **public/js/client.js**
  - Añadimos “server-message” a nuestro socket:

```
this.socket.on('server-message', ({ user, message }) => {
    this.addMessage(user, message);
});
```

  - Finalmente añadimos “connect” al socket:

```
this.socket.on('connect', () => {
    console.log('[INFO] Socket conectado');
    /*
     * Actualizar la interfaz gráfica
     */
    this.setUpEvents();
    this.registerCommands();
    this.addSystemMessage(
        'Use /ayuda para ver los comandos disponibles.'
    );
});
```

Node.js - Chat App Cliente

- En **public/js/client.js**
  - En nuestro metodo setupEvents:

```
setUpEvents = () => {
    this.messageInput.addEventListener('keypress', e => {
        let key = e.key || e.which || e.keyCode;
        if (key === 'Enter' || key === 13)
            this.sendMessage(this.messageInput.value);
    });

    this.sendBtn.addEventListener('click', () => {
        const message = this.messageInput.value;
        this.sendMessage(message);
    });
};
```

- Donde unicamente enviamos el mensaje del input siempre y cuando en el input hayamos presionado enter o hayamos hecho click en el botón de enviar.

Node.js - Chat App Cliente

- En **public/js/client.js**
  - En nuestro método sendMessage:

```
sendMessage = message => {
    const minified = message.toLowerCase().trim();
    if (minified === '/ayuda' || minified === '/help') this.printCommands();
    else if (minified === '/ping') this.emit('ping');
    else if (minified === '/salir' || minified === '/exit') {
        this.isWaitingClose = true;
        this.emit('close');
    } else if (minified.length > 0) {
        this.emit('client-message', { message: message });
        this.addMessage(this.user, message, true);
    }

    this.messageInput.value = '';
};
```

- Donde filtramos comandos de nuestro mensaje y limpiamos el input.

Node.js - Chat App Cliente

- En `public/js/client.js`
  - En nuestro método emit:

```
emit = (command, payload = {}) => {
    if (!Boolean(this.socket)) throw new Error('Socket not initialized');
    this.socket.emit(command, payload);
};
```

  - En nuestro método close:

```
close = () => {
    if (!Boolean(this.socket)) throw new Error('Socket not initialized');
    this.socket.emit('close');
};
```

- Ver demostración en vivo en Chat App.
- Ver repositorio para más detalles.

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

Ejercicios de práctica

Mision TIC 2022

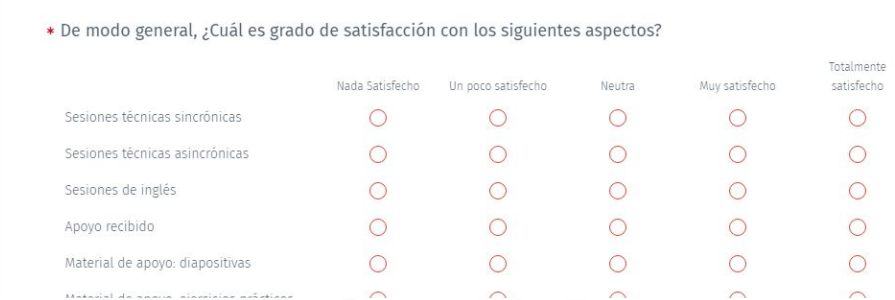
Referencias

- https://nodejs.org/en/knowledge/HTTP/servers/how-to-create-a-HTTP-server/
- https://devqa.io/nodejs-server-example/
- https://expressjs.com/
- https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
- https://xanderbakx.medium.com/what-are-websockets-socket-io-e327c797b8a

El futuro digital es de todos | MinTIC

UNIVERSIDAD DEL NORTE
Vigilada Mineducación

Seguimiento Habilidades Digitales en Programación



* De modo general, ¿Cuál es grado de satisfacción con los siguientes aspectos?

	Nada Satisfecho	Un poco satisfecho	Neutra	Muy satisfecho	Totalmente satisfecho
Sesiones técnicas sincrónicas	○	○	○	○	○
Sesiones técnicas asincrónicas	○	○	○	○	○
Sesiones de inglés	○	○	○	○	○
Apoyo recibido	○	○	○	○	○
Material de apoyo: diapositivas	○	○	○	○	○

Completa la siguiente encuesta para darnos retroalimentación sobre esta semana ▼▼▼

https://www.questionpro.com/t/ALw8TZlxOJ

MisionTIC2022

El futuro digital es de todos

MinTIC

UNIVERSIDAD DEL NORTE

Vigilada Mineducación

¡GRACIAS

POR SER PARTE DE ESTA EXPERIENCIA DE APRENDIZAJE!

Mision TIC 2022