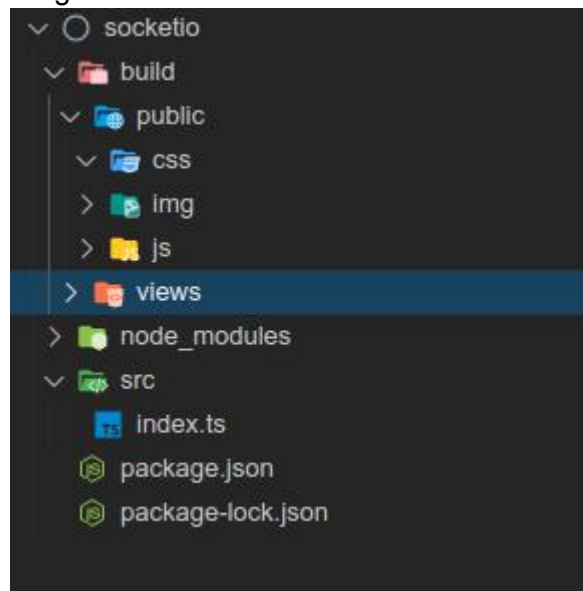


Requisitos para el siguiente tutorial

- ✓ Conocimiento en POO
- ✓ Conocimiento en JavaScript
- ✓ Conocimientos básicos en TypeScript
- ✓ Conocimientos básicos en Node JS
- ✓ Conocimientos en HTML
- ✓ Se recomienda conocimientos en el framework css bootstrap

PASOS A SEGUIR:

1. Se crea una carpeta para los archivos del proyecto.
2. Se abre una consola en la ubicación del proyecto.
3. Se ejecuta el comando “npm init –yes”.
4. Abrir el proyecto con su editor de código favorito. Para este ejemplo se usará visual studio code.
5. Se procede a crear la siguiente estructura:



- ✓ La carpeta build es donde se situará el código transpilado para que Node Js pueda correr el servidor.
- ✓ En la carpeta src se procederá a crear toda la estructura del proyecto con código TypeScript.
- ✓ El archivo index.ts es el archivo de arranque de nuestro servidor, aquí se ubicará toda la lógica del sistema.
- ✓ La carpeta views contiene los archivos “.ejs” que son las plantillas que se van a renderizar al momento de solicitarlas al servidor.
- ✓ La carpeta public que contendrá cualquier archivo estático que se quiera mostrar en la web renderizada.

6. Luego, se procede a configurar el archivo `tsconfig.json`, que es el archivo de configuración del proceso de transpilación. No olvidar que para dar paso a la creación de dicho archivo debemos tener instalado TypeScript¹ y usar el comando `"tsc -init"`.

```
socketio$ tsc -init
message TS6071: Successfully created a tsconfig.json file.
```

7. A continuación, se configura el archivo `tsconfig.json` los siguiente.

```
"target": "es6",
"outDir": "./build",
```

8. Luego de esto, se instala `express` y otros módulos necesarios para crear el servidor con el comando `"npm i express morgan ejs socket.io"` y como dependencia de desarrollo con el comando `"npm i -D @types/express @types/morgan nodemon"`.
9. Después, se debe configurar el controller del servidor.

```
ketio > src > controller > indexController.ts > indexController > cargarIndex
import { Request, Response } from "express";
class indexController {
  public cargarIndex(req: Request, res: Response) {
    res.render("index")
  }
}

export default new indexController();
```

En este caso se procede a crear solo un método que muestre la vista que contiene el template que usaremos para el tutorial, sera algo sencillo con fines demostrativos solo se creara una simple vista de un chat.

¹ [TypeScript](#)

10. Se procede a configurar el archivo de router que solo tendrá una para mostrar la vista que hemos creado.

```
ketio > src > router > indexRouter.ts > routerIndex > config
import { Router } from "express";
import indexController from "../../controller/indexController"
class routerIndex {
  public router: Router = Router();

  constructor() {
    this.config();
  }

  config():void {
    this.router.get("/", indexController.cargarIndex);
  }
}
export default new routerIndex().router
```

11. Lo siguiente es configurar el archivo principal.
12. Se realizan los imports.

```
ketio > src > index.ts > Server
import express, { Application } from "express";
import morgan from "morgan";
const bodyParser = require("body-parser");
const path = require("path");
const SocketIO = require('socket.io');
```

- I. Express y Application corresponden a los módulos requeridos para crear el servidor en express.
 - II. Con Morgan se procede a configurar una interfaz para mostrar por consola las rutas a las que se acceden.
 - III. bodyParser es para poder transmitir JSON por los métodos get, post, etc.
 - IV. Finalmente, el módulo Path es para trabajo interno de rutas.
 - V. SockerIO es la librería que va a permitir crear el servidor de sockets para la practica.
13. Luego se procede a crear las variables para almacenar los datos del chat y realizar la emisión de los mensajes por medio de sockets.

```
var io:any;
const mensajes: any[] = [
];
const usuarios: any[] = [
]
```

Son 3 constantes, la constante **io** para gestionar los sockets y la constante de **mensajes y usuarios** para almacenar la información del chat.

14. Para poder empezar con el desarrollo de nuestro servidor se requiere de una clase que represente a nuestro servidor, la cual será la siguiente:

```
class Server {  
  public app:Application;  
}
```

15. Luego de esto, se crea el método de configuración del servidor.

```
config(): void {  
  this.app.set("port", process.env.PORT || 3000);  
  //static files  
  this.app.set('views', path.join(__dirname, 'views'));  
  this.app.set('view engine', 'ejs');  
  this.app.use(express.static(path.join(__dirname, '/public')));  
  this.app.use(morgan("dev"));  
  this.app.use(bodyParser.json());  
  this.app.use(bodyParser.urlencoded({ extended: true }));  
}
```

Como se puede observar, tenemos que configurar un puerto en el cual nuestro servidor estará escuchando las peticiones a las rutas configuradas y las solicitudes de sockets.

```
this.app.set("port", process.env.PORT || 3000);
```

Cuando se desarrolla en modo local podemos especificar el puerto que más nos convenga, pero cuando se realiza el despliegue de nuestra api en algún VPS (Segarra, 2019) como heroku, por ejemplo, estos servidores nos proporcionan un puerto en concreto por lo cual se accede a este por medio de la variable de entorno (Aosbot, 2019) “**process.env.PORT**”.

Si nuestro servidor devuelve algún tipo de archivo estático, necesitamos configurar un directorio de acceso público para que se pueda guardar este tipo de información aquí, lo cual se realiza con:

```
this.app.use(express.static(path.join(__dirname, '/public')));
```

El “/public” representa la carpeta donde se almacenarán los archivos estáticos, dicha carpeta debe estar ubicada en el directorio “build”, ya que aquí es donde se mantendrá la información que será visible para el consumidor de nuestro servidor, se incluirán los archivos .js para la configuración de sockets y cualquier otro archivo estático para mostrar la web.

```
this.app.use(morgan("dev"));
```

Con esta línea se indica a express que utilice el modulo **Morgan**² el cual recibe un Sting como parámetro entrada, con lo cual podremos observar de manera resumida las rutas que son accedidas en nuestro servidor.

```
this.app.use(bodyParser.json());
```

² [Documentación de Morgan](#)

Con esta configuración le indicamos a express que puede recibir un json en la comunicación del servidor.

```
this.app.use(bodyParser.urlencoded({ extended: true }));
```

Y en la última configuración, lo que le indicamos a nuestro servidor es que solo recibirá por la url datos planos, ya que al momento no se transmiten archivos por la url en nuestro servidor.

Cabe recalcar que estos últimos dos pasos no son necesarios ya que no realizaremos comunicación por medio de url pero siempre es recomendable tener esta configuración para evitar cualquier error o por el hecho de que la aplicación pueda crecer.

16. Una vez configurado nuestro servidor debemos asignarle las rutas permitidas de acceso

```
router(): void {  
  this.app.use("/", routerIndex);  
}
```

La asignación de rutas cuenta de dos partes, la primera que es la ruta base desde la cual se empezará a trabajar y nuestro archivo donde configuramos las rutas que estarán disponibles en dicha ruta base.

Adicional necesitamos configurar nuestro servidor de sockets, para realizar esto nos valdremos de una función adicional con la cual realizaremos toda la estructura de la comunicación de mensajes a través de los sockets.

```

configSocketServer() {
  io = SocketIO.listen(this.server)
  io.on('connection', (socket: any) => {
    console.log("Nueva conexión de socket ID: " + socket.id);
    socket.on("iniciar", function (data: any) {
      usuarios.push(
        {
          user: data.nombre,
          foto: data.foto,
          id: socket.id
        }
      );
      io.emit("conectados", usuarios);
      io.emit("entrada", mensajes);
    });

    socket.on('disconnect', function () {
      console.log("Se desconecto el usuario: " + socket.id);
    });

    socket.on('mensajeEnviado', function (data: any) {
      mensajes.push(data);
      io.emit("entrada", mensajes);
    });
  });
}

```

Como observamos en la imagen, cada conexión de socket tiene a su disposición todos estos métodos, cuando se realiza una conexión entrante se configura todos estos estados de sockets, adicional se puede realizar cualquier lógica que se requiera para el sistema en este caso es solo un console log que indica que se realizó la conexión.

El evento **on** indica que es una conexión entrante, es decir el servidor recibe una conexión entrante que se la identifica con el nombre que ponemos como primer argumento y la procesa usando el callback que se asocia como segundo argumento.

Este callback puede tener la lógica que uno requiera en nuestro caso guardaran los usuarios, mensajes e iniciaran la configuración del chat.

Observamos que hay dos eventos **on** especiales uno que representa la conexión nueva y uno que representa la desconexión del socket.

Caso contrario para emitir notificaciones en los sockets es decir enviar eventos de salida se debe usar la constante **io** que es donde esta escuchando el servidor de sockets y llamamos al método emit cuyo primer parámetro es el identificador del mensaje y su segundo parámetro son los datos a enviar que pueden ser de cualquier tipo.

17. Finalmente, para terminar con la programación de nuestro servidor, debemos realizar el método que levantará nuestro servidor.


```

start(): void {
  this.server = this.app.listen(this.app.get("port"), () => {
    console.log("server on port: ", this.app.get("port"));
  });
  this.configSocketServer();
}

```

El método start lo único que realizará es indicarle a nuestro servidor que se ponga a escuchar peticiones en un puerto específico, el mismo que fue configurado previamente y se lo puede acceder mediante el comando “**this.app.get(‘port’)**”, como esto requiere un callback se hará mediante la implementación de una función flecha que lo único que hará es indicar que el servidor escucha en “X” puerto.

Adicional se hace llamado a la función de configuración de sockets para que se levante el servidor socket para aceptar peticiones de este tipo y que funcione el chat.

18. Métodos adicionales de configuración.

```

constructor() {
  this.app = express();
  this.config();
  this.router(); }

```

No olvidemos que todos estos son métodos de la clase Server y que para poder usarlos debemos crear una instancia de la misma y llamar a su método start que es el encargado de arrancar todo nuestro servidor.

```

const server = new Server();
server.start();

```

Las configuraciones de la vista y css no se mostraran en este tutorial ya que no afectan en nada al servidor esto queda a la creatividad de cada uno, lo que si necesitamos saber es como configurar el cliente socket en el archivo .js que se vincula en el index y este archivo debe estar en la carpeta public del servidor.

```

var socket = io.connect('http://localhost:3000');
var nombre = "";
var foto = "";

document.addEventListener("DOMContentLoaded", function(event) {
  nombre = prompt("Ingrese su nombre");
  foto = prompt("Link de foto", "https://www.creartuavatar.com/images/m7.svg");
  socket.emit("iniciar", { nombre: nombre, foto: foto });
});

socket.on('entrada', (data) => {
  render(data)
});

socket.on("conectados", (data) => {
  renderUsuarios(data)
});

function render(data) {
  var html = data.map(function (elem, index) {
    return (`<div>
      <strong>${elem.author}</strong>:
      <em>${elem.text}</em>
    </div>`)
  }).join(" ");
  document.getElementById('messages').innerHTML = html;
  var element = document.createElement("div");
  element.id = "foco";
  document.getElementById('messages').parentNode.append(element);
  sinjQuery();
}

```



```

function renderUsuarios(data) {
  var html = data.map(function (elem, index) {
    return (
      <li class="list-group-item p-0 bordePersonalizado">
        <div class="row">
          <div class="col-sm-4">
            
          </div>
          <div class="col-sm-8 d-flex align-middle">
            <p class="text-justify centrado text-wrap">
              {elem.user}
              <br>
              ID: {elem.id}
            </p>
          </div>
        </div>
      </li>
    )
  }).join("");
  document.getElementById("usuarios").innerHTML = html;
}

function enviarMensaje() {
  var mensaje = document.getElementById("inputMsg").value;
  socket.emit("mensajeEnviado", { author: nombre, text: mensaje });
  document.getElementById("inputMsg").value = "";
}

function sinjQuery() {
  var e = document.getElementById('scroll');
  e.innerHTML += '<div class="chatMessage"></div>';
  var objDiv = document.getElementById("scroll");
  objDiv.scrollTop = objDiv.scrollHeight;
}

```

Como observamos en el archivo la configuración es sencilla, necesitamos crear una variable socket que será la encargada de gestionar el web socket cliente, para crear esto llamamos a la instancia **io** y a su método **connect** que recibe como entrada la url donde se está ejecutando nuestro servidor de socket, esta variable **io** se obtiene de la importación del archivo **/socket.io/socket.io.js** esto se lo hace mediante la etiqueta script como si se tratase de cualquier archivo **.js** esto es gracias a que nuestro servidor socket fue configurado previamente y automáticamente da acceso a esta ruta pero si fuese en un servidor independiente se debería de hacer la instalación de la librería de socket.io respectiva para el cliente que se este usando.

Los métodos de comunicación de los sockets son opuestos a los que se tienen en el servidor es decir si en el servidor está con **on** en el cliente estará con **emit** y viceversa.

Hasta el momento ya tenemos configurado nuestro servidor, pero ¿cómo lo ponemos en ejecución?. Sencillamente, debemos configurar el archivo de arranque de Node JS para que así podamos iniciar nuestra app.

Para lograr esto debemos dirigirnos al archivo **"package.json"** de nuestra carpeta principal y crear los scripts necesarios para transpilar y otro para arrancar nuestro servidor.

19. El script para transpilar nuestro código.

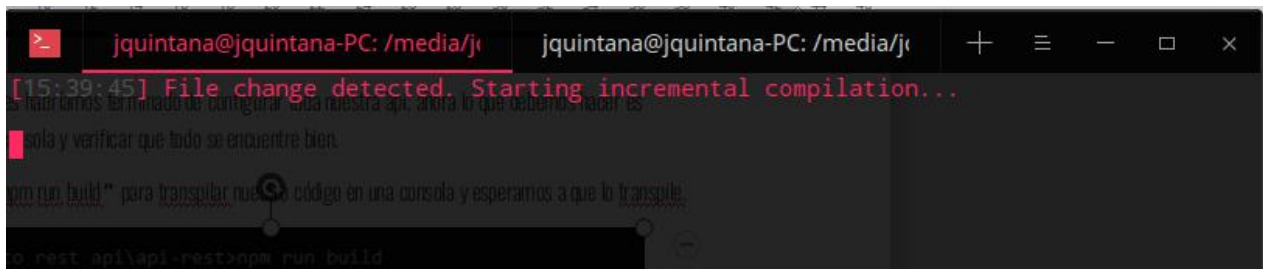
```
"build": "tsc -w",
```

20. Mientras que el script para arrancar el servidor.

```
"start": "node build/index.js"
```

Con estos últimos ajustes habríamos terminado de configurar toda nuestro servidor, ahora lo que debemos hacer es ejecutarla mediante la consola y verificar que todo se encuentre bien.

Primero, ejecutamos **"npm run build"** para transpilar nuestro código en una consola y esperamos a que lo transpile.



The screenshot shows a terminal window with the title bar "jquintana@jquintana-PC: /media/jq". The terminal output displays a red message: "[15:39:45] File change detected. Starting incremental compilation...". Below this, there is a faint, partially visible command prompt showing "npm run build" and "o rest api/api-rest>npm run build".

Con el código ya transpilado, ahora debemos proceder a ejecutar nuestro servidor mediante el comando “**npm start**”.

```
[nodemon] restarting due to changes...  
[nodemon] starting `node build/index.js`  
server on port: 3000  
ver on port: ", this.app.get("port"));
```

Como podemos observar, nos indica que nuestro servidor está levantado en el puerto 3000, mismo puerto que configuramos previamente, para poder verificar que esté funcionando sin errores abrimos nuestro navegador e ingresamos a la siguiente dirección “<http://localhost:3000/>” debemos obtener el siguiente mensaje en el navegador



Y en la consola podemos observar que ruta fue solicitada.

```
GET / 304 9.679 ms - -  
Se desconecto el usuario: H4Metelh9RVMSiraAAAA  
Nueva conexion de socket ID: ilxcERw7AOvcYXhFAAAC
```

Si se va a estar en desarrollo del servidor es tedioso estar pausando y ejecutando a cada momento el comando “npm start” por lo cual se sugiere el uso del middleware nodemon³. Para que este proceso sea automático mediante el uso del comando **nodemon build/index.js** o si se configura en el package.json el comando “**npm run dev**”.

³ [Página oficial nodemon](#)

Referencias

Aosbot. (22 de Octubre de 2019). *Wikipedia*. Obtenido de Variable de entorno:
https://es.wikipedia.org/wiki/Variable_de_entorno

Segarra, F. (30 de Agosto de 2019). *¿Qué es un VPS? Todo lo que necesitas saber sobre servidores virtuales*. Obtenido de Hostinger: <https://www.hostinger.es/tutoriales/que-es-un-vps>

Repositorio git.

<https://github.com/jquintanas/socketio>