

## EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: PERSISTENCIA DE DATOS.
- EXERCISE 2: PERSISTENCIA DE DATOS II.
- EXERCISE 3: UTILIZANDO EXPRESS PARA CREAR UN SERVIDOR Y PARA MANEJO DE RUTAS DE ARCHIVOS ESTÁTICOS.
- EXERCISE 4: UTILIZANDO EXPRESS CON MOTOR DE PLANTILLA HANDLEBARS Y BOOTSTRAP.

## EXERCISE 1: PERSISTENCIA DE DATOS.

Para continuar con la práctica de la persistencia de datos en archivos de texto, utilizando Node js, armaremos un nuevo programa, el cual ingresará los datos de un color, y qué tanto te gusta en una escala numérica.

Crearemos tres archivos: escritura.js, lectura.js, y datos.txt

Dentro de nuestro archivo lectura.js, utilizaremos el método `readFile()` del módulo `fs`. Primero requerimos el módulo.

```
JS lectura.js  X
JS lectura.js > ...
1  const fs = require('fs/promises');
```

Hacemos uso del método `readFile()` para leer nuestro archivo de datos, y ya que nuestra información está en formato `JSON`, emplearemos `JSON.parse`.

```
JS lectura.js x
JS lectura.js > leerArchivo
1  const fs = require('fs/promises');
2
3  const leerArchivo = async () => {
4    const datos = await fs.readFile('datos.txt');
5    console.log(JSON.parse(datos));
6  }
7
8  leerArchivo()
```

Debido a que nuestro archivo se encuentra vacío, obtendremos el siguiente error.

```
C:\Users\persistencia-ejercicios>node lectura.js
(node:7292) UnhandledPromiseRejectionWarning: SyntaxError: Unexpected end of JSON input
    at JSON.parse (<anonymous>)
    at leerArchivo (C:\Users\persistencia-ejercicios\lectura.js:5:22)
    (Use `node --trace-warnings ...` to show where the warning was created)
(node:7292) UnhandledPromiseRejectionWarning: Unhandled promise rejection. This error originated either
    a catch block, or by rejecting a promise which was not handled with .catch(). To terminate the node p
    I flag `--unhandled-rejections=strict` (see https://nodejs.org/api/cli.html#cli_unhandled_rejections_m
    (node:7292) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecated. In the future,
    inate the Node.js process with a non-zero exit code.

C:\Users\persistencia-ejercicios>
```

Para este caso, debemos agregar una validación que indique cuando nuestro archivo esté vacío.

```

JS lectura.js x
JS lectura.js > ...
1  const fs = require('fs/promises');
2
3  const leerArchivo = async () => {
4      const datos = await fs.readFile('datos.txt');
5
6      if(datos.length == 0){
7          return console.log("El archivo se encuentra vacio");
8      }
9
10     console.log(JSON.parse(datos));
11 }
12
13 leerArchivo()
  
```

También es importante controlar los errores, y enviar mensajes útiles al usuario en caso de que éstos ocurran.

```

JS lectura.js x
JS lectura.js > ...
1  const fs = require('fs/promises');
2
3  const leerArchivo = async () => {
4      try {
5          const datos = await fs.readFile('datos.txt');
6
7          if(datos.length == 0){
8              return console.log("El archivo se encuentra vacio");
9          }
10
11         console.log(JSON.parse(datos));
12     } catch (error) {
13         console.log("Lo sentimos, ha ocurrido un error");
14         console.log(error);
15     }
16 }
17
18 leerArchivo()
  
```

PROBLEMS
OUTPUT
TERMINAL
DEBUG CONSOLE

```

C:\Users\persistencia-ejercicios>node lectura.js
El archivo se encuentra vacio

C:\Users\persistencia-ejercicios>
          
```

En nuestro archivo **escritura.js**, requerimos el módulo **fs**.

```
JS escritura.js X  
JS escritura.js > ...  
1  const fs = require('fs/promises');
```

Definiremos los parámetros de entrada por la línea de comandos. Empezaremos eliminando los dos primeros ítems, los cuales no usaremos; y luego, definiremos en una variable el color, y en otra su puntaje.

```
JS escritura.js X  
JS escritura.js > ...  
1  const fs = require('fs/promises');  
2  const argumentosEntrada = process.argv.slice(2);  
3  const color = argumentosEntrada[0];  
4  const puntaje = argumentosEntrada[1];
```

Antes de modificar la información dentro de nuestro archivo de texto, primero debemos obtener dicha información contenida en éste, para así poder agregar las nuevas propiedades. En una primera instancia, nuestro archivo de texto se encontrará vacío. Debido a estas condiciones, será de mucha ayuda realizar una verificación del archivo, y si es que éste se encuentra vacío, definiremos una variable con valor de objeto en vacío; y en caso contrario, la variable tomará el valor del objeto contenido en el archivo de datos.

```
JS escritura.js X
JS escritura.js > ...
1  const fs = require('fs/promises');
2  const argumentosEntrada = process.argv.slice(2);
3  const color = argumentosEntrada[0];
4  const puntaje = argumentosEntrada[1];
5
6  let objetoDatos = {};
7
8  const obtenerDatos = async () => {
9    try {
10     const datos = await fs.readFile('datos.txt')
11
12     if(datos.length !== 0){
13       objetoDatos = JSON.parse(datos);
14     }
15   } catch (error) {
16     console.log('Lo sentimos, ha ocurrido un error');
17     console.log(error)
18   }
19 }
20
21
22 obtenerDatos();
23
```

Ahora que ya tenemos nuestro objeto inicial, agregaremos los datos ingresados por la línea de comandos usando el **spread operator**, y pasando el nuevo objeto (primero a cadena de texto, con el método **JSON.stringify**) al método **writeFile()**.

```
JS escritura.js X
JS escritura.js > ...
1  const fs = require('fs/promises');
2  const argumentosEntrada = process.argv.slice(2);
3  const color = argumentosEntrada[0];
4  const puntaje = argumentosEntrada[1];
5
6  let objetoDatos = {};
7
8  const obtenerDatos = async () => {
9    try {
10     const datos = await fs.readFile('datos.txt')
11
12     if(datos.length !== 0){
13       objetoDatos = JSON.parse(datos);
14     }
15
16     const nuevoObjeto = {...objetoDatos, [color]: puntaje}
17
18     await fs.writeFile('datos.txt', JSON.stringify(nuevoObjeto, null, 2))
19
20     console.log("Los datos han sido agregados exitosamente")
21   } catch (error) {
22     console.log('Lo sentimos, ha ocurrido un error');
23     console.log(error)
24   }
25 }
26
27
28 obtenerDatos();
29
```

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node escritura.js azul 12
Los datos han sido agregados exitosamente

C:\Users\persistencia-ejercicios>
```

Comprobamos que nuestro archivo de texto efectivamente contiene los datos que le hemos pasado.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node escritura.js azul 12
Los datos han sido agregados exitosamente

C:\Users\persistencia-ejercicios>node lectura.js
{ azul: '12' }

C:\Users\persistencia-ejercicios>
```

Tener un archivo lectura.js para leer nuestro archivo programáticamente parece un poco ineficiente, considerando que dentro de éste ya estamos leyendo la información del archivo de texto, para luego agregar los datos modificados.

Para unificar estas tareas, podemos pasar un argumento a la línea de comandos, y que sea identificado como una opción al momento de ejecutar el archivo escritura.js.

Existen varias formas de hacerlo, pero ahora solo agregaremos una validación simple, en la que nuestro programa, al momento de leer el primer argumento, identifique la opción “leer”, y solo entonces realice la lectura de nuestro archivo en vez de agregar más datos.

Primero copiaremos la función `leerArchivo()` de nuestro archivo `lectura.js`, a nuestro archivo `escritura.js`.



```
JS escritura.js x JS lectura.js
JS escritura.js > ...
1  const fs = require('fs/promises');
2  const argumentosEntrada = process.argv.slice(2);
3  const color = argumentosEntrada[0];
4  const puntaje = argumentosEntrada[1];
5
6  let objetoDatos = {};
7
8  const leerArchivo = async () => {
9    try {
10      const datos = await fs.readFile('datos.txt');
11
12      if(datos.length == 0){
13        return console.log("El archivo se encuentra vacio");
14      }
15
16      console.log(JSON.parse(datos));
17    } catch (error) {
18      console.log("Lo sentimos, ha ocurrido un error");
19      console.log(error);
20    }
21  }
22
23
```

Y ya que nuestra función “obtener datos” no solo está obteniendo datos, sino que también los escribe en el archivo, lo más recomendable es cambiarle el nombre para hacerla más descriptiva (recuerda cambiar tanto la definición, como la llamada de la función).

```

JS escritura.js X JS lectura.js
JS escritura.js > ...
22
23
24 const escribirDatos = async () => {
25   try {
26     const datos = await fs.readFile('datos.txt')
27
28     if(datos.length !== 0){
29       | objetoDatos = JSON.parse(datos);
30     }
31
32     const nuevoObjeto = {...objetoDatos, [color]: puntaje}
33
34     await fs.writeFile('datos.txt', JSON.stringify(nuevoObjeto,null, 2))
35
36     console.log("Los datos han sido agregados exitosamente")
37   } catch (error) {
38     console.log('Lo sentimos, ha ocurrido un error');
39     console.log(error)
40   }
41 }
42
43
44 escribirDatos();
45
46
47
  
```

Ahora crearemos una nueva función que verifique los argumentos de entrada, buscando coincidencias con la palabra "leer".

```

43
44 const verificarOpcionEntrada = () => {
45   | if(argumentosEntrada[0] == 'leer'){
46     | return leerArchivo();
47   } else {
48     | return escribirDatos();
49   }
50 }
51
52 verificarOpcionEntrada();
53
  
```

Esto nos permite dividir nuestro programa en dos partes.



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node escritura.js leer
{ azul: '12' }

C:\Users\persistencia-ejercicios>node escritura.js verde 0
Los datos han sido agregados exitosamente

C:\Users\persistencia-ejercicios>node escritura.js leer
{ azul: '12', verde: '0' }

C:\Users\persistencia-ejercicios>
```

También podemos utilizar un bloque de tipo case, a cambio de un bloque **if**, de la siguiente forma.

```
52
53  const verificarOpcionEntrada = () => {
54    switch(argumentosEntrada[0]) {
55      case 'leer':
56        leerArchivo();
57        break;
58      default:
59        escribirDatos();
60    }
61  }
62
63  verificarOpcionEntrada();
64
```

Por ahora, si es que alguien llega a ejecutar nuestro programa sin ningún argumento, éste no leerá el archivo, y tampoco agregará nueva información al archivo de datos, a pesar de que el mensaje que recibimos es “Los datos han sido agregados exitosamente”; por lo tanto, un usuario que no se encuentre familiarizado con el programa, no sabrá que está pasando.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node escritura.js
Los datos han sido agregados exitosamente

C:\Users\persistencia-ejercicios>
```

Es por ello que realizar una validación a los datos es una buena práctica de entrada, para así asegurarnos que nuestro programa se ejecute según lo esperado, y también que el usuario reciba información útil respecto a cómo debe ser ejecutado.

```
JS escritura.js X
JS escritura.js > ...
52
53 const verificaParametrosEntrada = () => {
54   if(argumentosEntrada.length == 0){
55     console.log("Debes ingresar un color y un puntaje para agregar o modificar datos en archivo datos.txt")
56     console.log("Tambien puedes pasar la opcion leer, para conocer el contenido del archivo datos.txt")
57     return process.exit();
58   }
59 }
60
61 }
62
63 verificaParametrosEntrada();
64
65 verificarOpcionEntrada();
66
67
```

Entonces, si es que volvemos a ejecutar nuestro programa sin argumentos, ahora obtendremos el siguiente mensaje.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node escritura.js
Debes ingresar un color y un puntaje para agregar o modificar datos en archivo datos.txt
Tambien puedes pasar la opcion leer, para conocer el contenido del archivo datos.txt

C:\Users\persistencia-ejercicios>
```

## EXERCISE 2: PERSISTENCIA DE DATOS II.

Ahora utilizaremos los mismos conceptos aplicados en el programa anterior. Crearemos un programa interactivo, donde modificaremos el objeto con información respecto al lenguaje de programación **JavaScript**, utilizado previamente en el Text Class.

Primero, vamos a definir las acciones que queremos realizar sobre el archivo de texto. Hasta el momento, podemos actualizar y agregar nuevas propiedades al objeto; y ahora también añadiremos la opción de remover una propiedad de un archivo.

Como ya tenemos definidas estas acciones, ordenaremos las secciones que tendrá nuestro programa:

- 1.- Requerir módulo **fs**.
- 2.- Limpiar argumentos de entrada.
- 3.- Definir variables iniciales con argumentos de entrada.
- 4.- Verificar argumentos de entrada, buscando coincidencia con alguna de las opciones.
- 5.- Definir acciones del programa, dependiendo de opción inicial.
- 6.- Validación de argumentos y datos de entrada, incluyendo mensajes para el usuario.

Pasos 1, 2 y 3:

```
JS programaInteractivo.js X
JS programaInteractivo.js > ...
1  const fs = require('fs/promises');
2  const argumentosEntrada = process.argv.slice(2);
3  const opcion = argumentosEntrada[0];
4  const propiedad = argumentosEntrada[1];
5  const valor = argumentosEntrada[2];
6
```

Para el paso número 4, crearemos una función que evalúe las opciones de entrada.

```
const validarOpcionesEntrada = () => {
  switch(opcion){
    case 'leer':
      //Codigo para leer el archivo
      break;
    case 'agregar':
      //Codigo para agregar o modificar propiedad
      break;
    case 'eliminar':
      //Codigo para eliminar propiedad
      break;
    default:
      //Codigo para informar al usuario
  }
}
```

Para el paso 5, definiremos cada método correspondiente para cada acción.

Para el método de lectura, utilizaremos el mismo código que en el programa anterior, definiendo también un objeto inicial.

```
JS programalInteractivo.js X
JS programalInteractivo.js > leerArchivo
1  const fs = require('fs/promises');
2  const argumentosEntrada = process.argv.slice(2);
3  const opcion = argumentosEntrada[0];
4  const propiedad = argumentosEntrada[1];
5  const valor = argumentosEntrada[2];
6
7  let objetoDatos = {};
8
9  const leerArchivo = async () => {
10   try {
11     const datos = await fs.readFile('datos.txt');
12
13     if(datos.length == 0){
14       return console.log("El archivo se encuentra vacio");
15     }
16
17     console.log(JSON.parse(datos));
18   } catch (error) {
19     console.log("Lo sentimos, ha ocurrido un error");
20     console.log(error);
21   }
22 }
23
```

Para la función de escritura, también reciclaremos el método anterior, **escribirDatos()**, y reemplazaremos las nuevas variables iniciales de nuestro programa.

```
const escribirDatos = async () => {  
  try {  
    const datos = await fs.readFile('datos.txt')  
  
    if(datos.length !== 0){  
      objetoDatos = JSON.parse(datos);  
    }  
  
    const nuevoObjeto = {...objetoDatos, [propiedad]: valor}  
  
    await fs.writeFile('datos.txt', JSON.stringify(nuevoObjeto,null, 2))  
  
    console.log("Los datos han sido agregados exitosamente")  
  } catch (error) {  
    console.log('Lo sentimos, ha ocurrido un error');  
    console.log(error)  
  }  
}
```

Y para la opción de eliminar una propiedad, crearemos un nuevo método. Dentro de éste hemos agregado una validación, para que solo cuando exista la propiedad en el objeto, esta sea eliminada; de lo contrario, enviamos un mensaje al usuario.

```
const eliminaPropiedad = async () => {  
  try {  
    const datos = await fs.readFile('datos.txt')  
  
    if(datos.length !== 0){  
      objetoDatos = JSON.parse(datos);  
    }  
  
    if(objetoDatos.hasOwnProperty(propiedad)){  
      delete objetoDatos[propiedad];  
    } else {  
      console.log(objetoDatos)  
      console.log(propiedad)  
      return console.log("Esta propiedad no existe");  
    }  
  
    await fs.writeFile('datos.txt', JSON.stringify(objetoDatos,null, 2))  
  
    console.log("Los datos han sido eliminados exitosamente")  
  } catch (error) {  
    console.log('Lo sentimos, ha ocurrido un error');  
    console.log(error)  
  }  
}
```

Ahora crearemos el último método para validar los argumentos de entrada, o enviar mensajes al usuario.

```
const verificaParametrosEntrada = () => {  
  if(argumentosEntrada.length == 0){  
    console.log("Porfavor ejecutar programa con alguna de las siguientes opciones")  
    console.log("Opcion 1: leer")  
    console.log("Opcion 2: agregar propiedad valor")  
    console.log("Opcion 2: eliminar propiedad")  
    return process.exit();  
  }  
}
```

Ya solo nos queda agregar nuestras funciones al bloque switch, dentro de la función **validarOpcionesEntrada()**.

```
const validarOpcionesEntrada = () => {  
  switch(opcion){  
    case 'leer':  
      //Codigo para leer el archivo  
      leerArchivo();  
      break;  
    case 'agregar':  
      //Codigo para agregar o modificar propiedad  
      escribirDatos();  
      break;  
    case 'eliminar':  
      //Codigo para eliminar propiedad  
      eliminaPropiedad();  
      break;  
    default:  
      //Codigo para informar al usuario  
      verificaParametrosEntrada()  
  }  
}  
  
validarOpcionesEntrada();
```

Para probar nuestro programa, partiremos copiando el siguiente texto en nuestro archivo **datos.txt**.



```
1 {  
2   "nombre": "JavaScript",  
3   "backend": "true",  
4   "frontend": "true",  
5   "OOP": "true",  
6   "HerramientaBackend": "Node JS"  
7 }
```

Y a continuación, probaremos cada una de las opciones habilitadas.

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

C:\Users\persistencia-ejercicios>node programaInteractivo.js
Porfavor ejecutar programa con alguna de las siguientes opciones
Opcion 1: leer
Opcion 2: agregar propiedad valor
Opcion 2: eliminar propiedad

C:\Users\persistencia-ejercicios>node programaInteractivo.js leer
{
  nombre: 'JavaScript',
  backend: 'true',
  frontend: 'true',
  OOP: 'true',
  HerramientaBackend: 'Node JS'
}

C:\Users\persistencia-ejercicios>node programaInteractivo.js agregar asincrono true
Los datos han sido agregados exitosamente

C:\Users\persistencia-ejercicios>node programaInteractivo.js leer
{
  nombre: 'JavaScript',
  backend: 'true',
  frontend: 'true',
  OOP: 'true',
  HerramientaBackend: 'Node JS',
  asincrono: 'true'
}

C:\Users\persistencia-ejercicios>node programaInteractivo.js eliminar asincrono
Los datos han sido eliminados exitosamente

C:\Users\persistencia-ejercicios>node programaInteractivo.js leer
{
  nombre: 'JavaScript',
  backend: 'true',
  frontend: 'true',
  OOP: 'true',
  HerramientaBackend: 'Node JS'
}

C:\Users\persistencia-ejercicios>
```

Copia del código programa interactivo (segundo ejercicio):



```
1 const fs = require('fs/promises');
2 const argumentosEntrada = process.argv.slice(2);
3 const opcion = argumentosEntrada[0];
4 const propiedad = argumentosEntrada[1];
5 const valor = argumentosEntrada[2];
6
7 let objetoDatos = {};
8
9 const leerArchivo = async () => {
10   try {
11     const datos = await fs.readFile('datos.txt');
12
13     if(datos.length == 0){
14       return console.log("El archivo se encuentra vacio");
15     }
16
17     console.log(JSON.parse(datos));
18   } catch (error) {
19     console.log("Lo sentimos, ha ocurrido un error");
20     console.log(error);
21   }
22 }
23
24 const escribirDatos = async () => {
25   try {
26     const datos = await fs.readFile('datos.txt')
27
28     if(datos.length !== 0){
29       objetoDatos = JSON.parse(datos);
30     }
31
32     const nuevoObjeto = {...objetoDatos, [propiedad]: valor}
33     await
34     fs.writeFile('datos.txt', JSON.stringify(nuevoObjeto, null, 2))
35
36     console.log("Los datos han sido agregados exitosamente")
37   } catch (error) {
38     console.log('Lo sentimos, ha ocurrido un error');
39     console.log(error)
40   }
41 }
42
43 const eliminaPropiedad = async () => {
44   try {
45     const datos = await fs.readFile('datos.txt')
46
47     if(datos.length !== 0){
48       objetoDatos = JSON.parse(datos);
49     }
50
51     if(objetoDatos.hasOwnProperty(propiedad)) {
52       delete objetoDatos[propiedad];
```





```
53     } else {
54         console.log(objetoDatos)
55         console.log(propiedad)
56         return console.log("Esta propiedad no existe");
57     }
58     await fs.writeFile('datos.txt',
59 JSON.stringify(objetoDatos,null, 2))
60
61     console.log("Los datos han sido eliminados exitosamente")
62
63     } catch (error) {
64         console.log('Lo sentimos, ha ocurrido un error');
65         console.log(error)
66     }
67 }
68
69 const verificaParametrosEntrada = () => {
70     if(argumentosEntrada.length == 0){
71         console.log("Porfavor ejecutar programa con alguna de las
72 siguientes opciones")
73         console.log("Opcion 1: leer")
74         console.log("Opcion 2: agregar propiedad valor")
75         console.log("Opcion 2: eliminar propiedad")
76         return process.exit();
77     }
78 }
79
80 const validarOpcionesEntrada = () => {
81     switch(opcion){
82         case 'leer':
83             //Codigo para leer el archivo
84             leerArchivo();
85             break;
86         case 'agregar':
87             //Codigo para agregar o modificar propiedad
88             escribirDatos();
89             break;
90         case 'eliminar':
91             //Codigo para eliminar propiedad
92             eliminaPropiedad();
93             break;
94         default:
95             //Codigo para informar al usuario
96             verificaParametrosEntrada()
97     }
98 }
99 validarOpcionesEntrada();
100
```

### EXERCISE 3: UTILIZANDO EXPRESS PARA CREAR UN SERVIDOR Y PARA MANEJO DE RUTAS DE ARCHIVOS ESTÁTICOS.

En este ejercicio profundizaremos más sobre **Express**. Al igual que con **Node**, con **Express** también tenemos la opción de crear un servidor HTTP de una manera más sencilla, y además, nos va a permitir gestionar las respuestas o solicitudes que hace el cliente por medio de diferentes verbos **HTTP**, como vimos en los CUEs anteriores.

**Express** también da la posibilidad de hacer sitios web dinámicos, conectarnos a bases de datos, y “pintar” toda esa información en un HTML enriquecido.

Lo primero que haremos será crear una nueva carpeta, en la que queramos tener el proyecto con la estructura base creada con **Node**. Como lo hemos estudiado en los CUEs anteriores, empezamos con el comando `npm init`, y podemos ver los archivos que se crean.

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess at reasonable defaults.

What do you want to name this package? > proyecto
What version of the package do you want to use? > 1.0.0
What description do you want to use? > 
What email address do you want to use to contact the project's maintainer? > 
What is the entry point of your package? > 
What keywords do you want to use to describe this project? > 
What is the license of your package? > MIT
package.json file created.

PROYECTO
├── node_modules
├── package-lock.json
└── package.json
```

Lo siguiente que haremos será instalar **Express**, y para ello utilizaremos el comando `npm install express`, fijándonos que estemos posicionados en la carpeta del proyecto que utilizaremos.

```
$ npm install express
npm notice created a lockfile for package-lock.json. You should not edit this file.
npm WARN proyecto@1.0.0 No description, no license, no test, no repository, no bugs, no homepage
npm WARN proyecto@1.0.0 No repository field.
```

Una vez terminada la instalación, crearemos el archivo `app.js`, y lo primero que escribiremos será “llamar a **express**”; para esto, generaremos una constante **express** y ésta requerirá a **express**.

```
js / ...  
const express = require("express");
```

A continuación, crearemos la constante `app` que utilizará a `express`, y configuraremos el puerto que generalmente es el `3000`.

```
const app = express();  
const port = 3000;
```

Utilizamos la constante `app`, la cual responderá a una solicitud que hará el cliente por medio del método `get`, que es el que generalmente utilizamos al ingresar una URL en nuestro navegador, es decir, le estamos solicitando al servidor ciertos archivos para visualizar la web que queremos visitar.

Después del método `get`, pediremos visualizar en la página raíz un requerimiento, junto a una respuesta; aquí haremos una función flecha, y la respuesta la enviaremos con un string que dirá `"Mi respuesta desde express"`. En resumen, estamos especificando que al momento de acceder a la ruta raíz ("/"), nosotros como servidor, vamos a responder con el `string` que definimos.

```
app.get("/", (req, res) => {  
  res.send("Mi repuesta desde express");  
});
```

Y, por último, tenemos que definir que nuestro servidor esté escuchando las peticiones que se harán. Para ello utilizaremos `app.listen`, y entre paréntesis, colocamos el puerto; luego, en una función flecha, colocaremos un mensaje con `console.log`, que dirá `"Servidor preparado en el puerto"`; y colocaremos el puerto representado por la constante `port`.

```
app.listen(port, () => {  
  console.log('Servidor preparado en el puerto', port);  
});
```

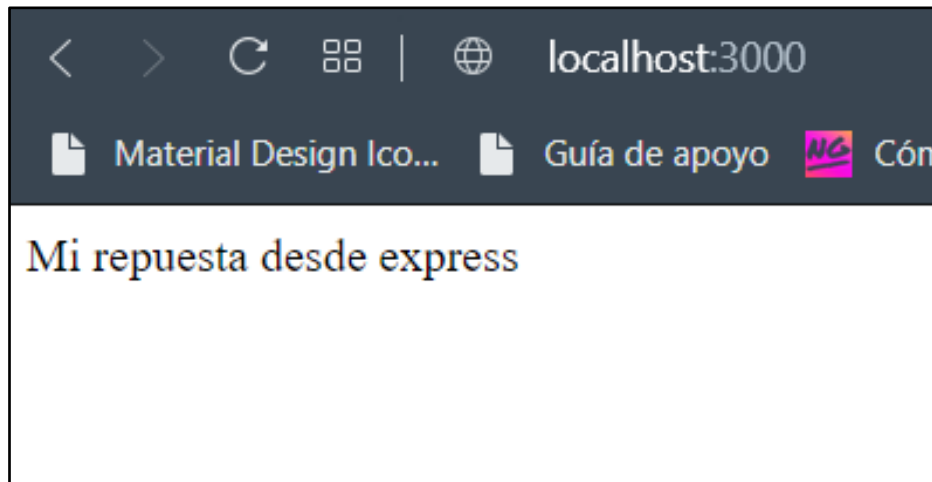
Ahora, en la consola de Visual Studio Code escribiremos el comando para levantar nuestro proyecto, y escribiremos **nodemon** junto al archivo que utilizaremos, en este caso será **app**. Presionamos enter.

```
$ nodemon app  
[nodemon] 2.0.15  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
Servidor preparado en el puerto 3000
```

Recordemos que para instalar **nodemon** de forma global, debemos utilizar el comando **npm install -g nodemon**.

En la consola nos aparecerá que el servidor estará a la escucha de los cambios que se realicen, y podremos ver el mensaje "Servidor preparado en el puerto 3000".

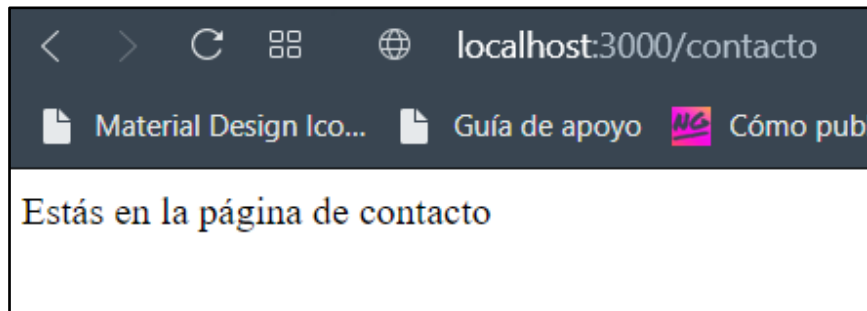
Para comprobar que todo está funcionando correctamente, nos vamos a dirigir a la URL **localhost:3000**, y podremos ver el mensaje que utilizamos "Mi respuesta desde Express".



Para continuar probando rutas, ahora podemos configurar otra del tipo `get`, la cual llamaremos `contacto`. Debajo de la ruta raíz escribiremos: `app.get`, y entre paréntesis colocaremos `/contacto`, junto a un requerimiento y una respuesta, también haremos una función flecha, y dentro ésta la respuesta será enviar un `string` que dirá `"Estás en la página de contacto"`.

```
app.get("/contacto", (req, res) => {  
  res.send("Estás en la página de contacto");  
});
```

Guardamos todos los cambios, y visualizamos nuestra nueva ruta configurada `localhost:3000/contacto`, donde podemos ver finalmente en el navegador el mensaje destinado a la vista de contacto. Con esto, podemos notar que ya estamos respondiendo a las solicitudes que hace el cliente; es decir, éste hace solicitudes a determinadas URL, y nosotros las respondemos, en este caso, con los mensajes que definimos para cada ruta.



Existen varias maneras de responder: con archivos **JSON** y con archivos estáticos. Para esto, se deberá crear una carpeta aparte que contenga todos los archivos estáticos que necesitemos para nuestra web, los cuales pueden ser: imágenes, HTML, CSS, y JavaScript.

Ahora pasaremos a ver los archivos estáticos. Lo primero que haremos será configurar una carpeta **public**; recordemos que los archivos que tenemos creados hasta el momento en nuestro proyecto corresponden únicamente a nuestro servidor, y el usuario tendrá acceso sólo a la carpeta que destinemos como pública.

Por lo tanto, crearemos la carpeta **public** en la raíz de nuestro proyecto, y en ella tendremos todos los archivos a los que el cliente podría acceder.

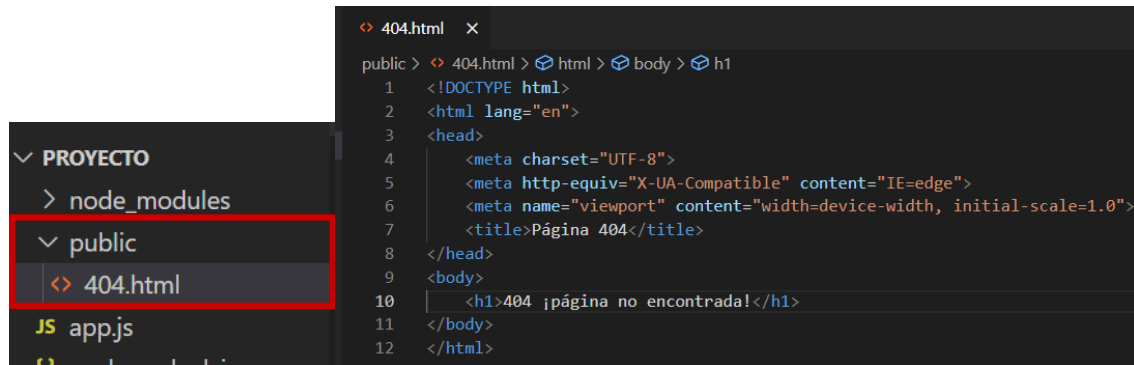
Lo siguiente que haremos será indicarle a **express** que ahora utilizaremos la carpeta que acabamos de crear. Después de la ruta de contacto, configuraremos nuestro primer **middleware**, escribiremos **app.use**, y entre paréntesis, emplearemos una función de Express llamada **static**.

Al utilizar **static**, tenemos que especificar la ruta donde estará nuestra carpeta. Para ello, a continuación de **static**, colocaremos **\_\_dirname** entre paréntesis, que hace alusión a la ruta que nosotros tendremos configurada, ya sea de archivos locales o nuestro servidor, y después un signo más, y entre comillas, la ruta de nuestra carpeta **/public**; por último, guardamos todos los cambios.

```
app.get("/contacto", (req, res) => {
  res.send("Estás en la página de contacto");
});

app.use(express.static(__dirname + "/public"))
```

Ahora crearemos nuestro primer archivo en la carpeta `public`, que se llamará `404.html`. Generaremos la estructura básica de un archivo `HTML`, como título pondremos `"Página 404"`, y en un `h1` pondremos `"404 ¡página no encontrada!"`.

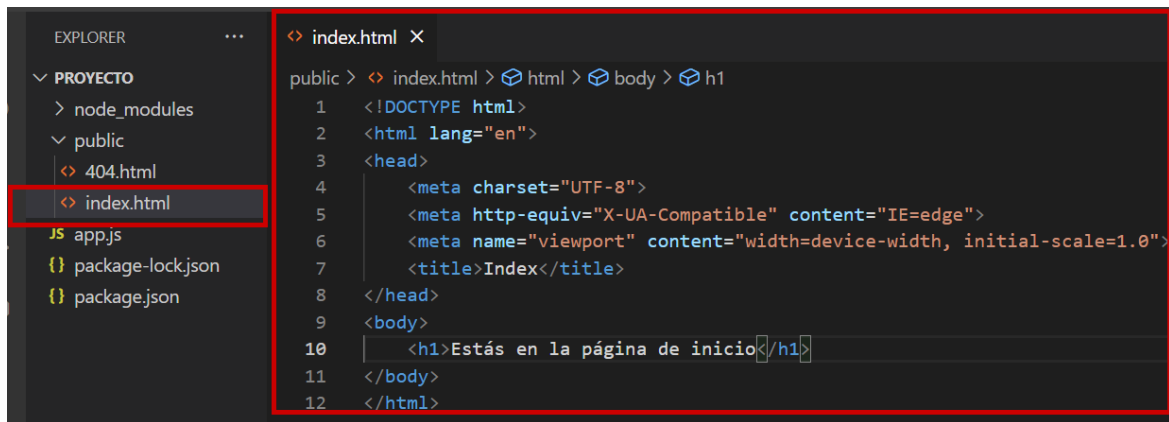


```
< 404.html x
public > < 404.html > html > body > h1
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Página 404</title>
8 </head>
9 <body>
10   <h1>404 ¡página no encontrada!</h1>
11 </body>
12 </html>
```

Guardaremos los cambios que realizamos. Para comprobar que todo está funcionando, vamos al navegador, y a la `URL localhost:3000` le agregaremos `/404.html`; al presionar enter podemos ver el contenido de ese archivo.



También haremos la prueba con otro archivo `HTML`, al cual llamaremos `index.html`, y que estará dentro de la carpeta `public`. Colocaremos la estructura básica de este tipo de archivos, como título pondremos `Index`, y agregaremos un `h1` que diga `"Estás es la página de inicio"`.



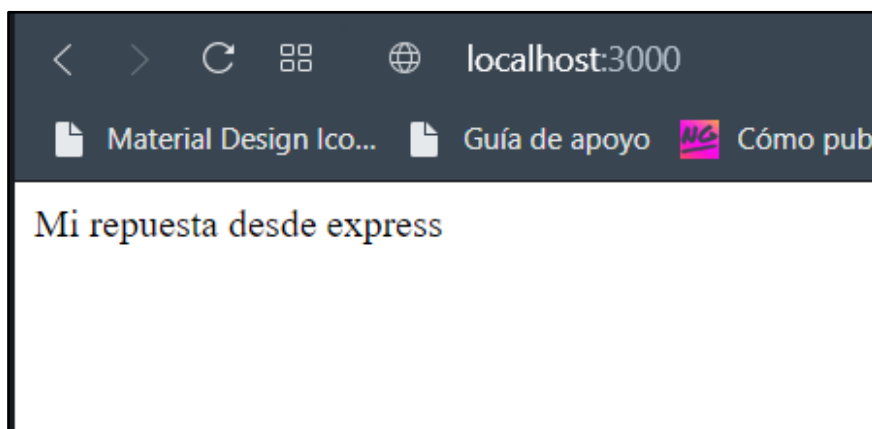
The screenshot shows the VS Code interface. On the left, the 'EXPLORER' sidebar displays a project structure with folders 'node\_modules' and 'public'. The 'public' folder is expanded, showing files '404.html', 'index.html' (highlighted with a red box), 'app.js', 'package-lock.json', and 'package.json'. The main editor area shows the content of 'index.html', which is a standard HTML5 boilerplate. The content is as follows:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Index</title>
8 </head>
9 <body>
10  <h1>Estás en la página de inicio</h1>
11 </body>
12 </html>

```

Guardamos, y nos dirigimos al navegador. Al colocar la URL `localhost:3000` (que sería nuestra ruta raíz), podremos notar que seguimos viendo el `string` definido en la ruta de nuestro archivo `app.js`.



Para cambiar esto, moveremos la línea de código perteneciente al `middleware` que creamos, y lo pondremos antes que se configure el `router`.



```

1  const express = require("express");
2
3  const app = express();
4  const port = 3000;
5
6
7  app.get("/", (req, res) => {
8    res.send("Mi respuesta desde express");
9  });
10
11 app.get("/contacto", (req, res) => {
12   res.send("Estás en la página de contacto");
13 });
14
15 app.use(express.static(__dirname + "/public"))
16
17
18 app.listen(port, () => {
19   console.log('Servidor preparado en el puerto', port);
20 });
21

```

Volvemos a guardar, actualizamos el navegador, y podemos ver finalmente el contenido del archivo **index.html**.



Tenemos dos formas de trabajar para ver nuestras vistas. La más rápida y sencilla es la que acabamos de ver, que sirve para crear páginas web tradicionales, es decir, no será necesario configurar los **get** para obtener las rutas, y ya tendremos en **public** la configuración de todas las páginas que utilizaremos, en el caso del ejemplo, las páginas: **404**, **index** y **contacto**.

Si revisamos los archivos **HTML** que creamos, podremos notar que tienen el mismo código, es decir, se repite la misma cabecera y el elemento **h1**. Para ahorrarnos el tener que escribir el mismo código varias veces, la segunda forma de trabajar las vistas, y que sería lo óptimo, es tener una carpeta que se llame **vistas** o **views**, donde será posible utilizar templates que

permiten hacer nuestro **HTML** dinámico gracias a **JavaScript**, y **Express** nos ayudará a leer esos archivos dinámicos y a transformarlos en nuestra carpeta **public**.

Para seguir practicando, probaremos otra manera de mostrar nuestra página 404, utilizando un nuevo tipo de middleware.

Lo que haremos ahora será escribir, a continuación del código donde apuntamos nuestra carpeta **public**, **"app.use"**, y entre paréntesis le pasaremos el requerimiento, la respuesta, y un **next**. Seguimos con una función flecha, y dentro tendremos la respuesta que va a ser el **status 404**, es decir, que cuando no se encuentre una página, nosotros enviaremos un archivo que estará alojado en el **\_\_dirname**, más la ruta con el archivo **404.html**.

```
app.use((req, res, next) =>{  
  res.status(404).sendFile(__dirname + "/public/404.html")  
})
```

Guardamos, visualizamos nuestro sitio web, y modificaremos la **URL**, escribiendo algo después de **localhost:3000**, en este caso pondremos **/game**, presionamos enter, y como resultado podemos ver el **h1** de la página 404 que creamos.



Ahora, si accedemos a **/contacto**, podemos ver que nos sigue respondiendo con 404.



Esto se debe a que tenemos establecido que nuestro middleware se ejecute antes que las rutas que habíamos definido, entonces cortaremos el código correspondiente al **middleware**, y lo pegaremos al final de las rutas definidas.

JS app.js

```

4   const port = 3000;
5
6   app.use(express.static(__dirname + "/public"))
7
8   app.use((req, res, next) => {
9     res.status(404).sendFile(__dirname + "/public/404.html")
10  })
11
12
13  app.get("/", (req, res) => {
14    res.send("Mi respuesta desde express");
15  });
16
17  app.get("/contacto", (req, res) => {
18    res.send("Estás en la página de contacto");
19  });
20
21
22
23  app.listen(port, () => {
24    console.log('Servidor preparado en el puerto', port);
25  });

```

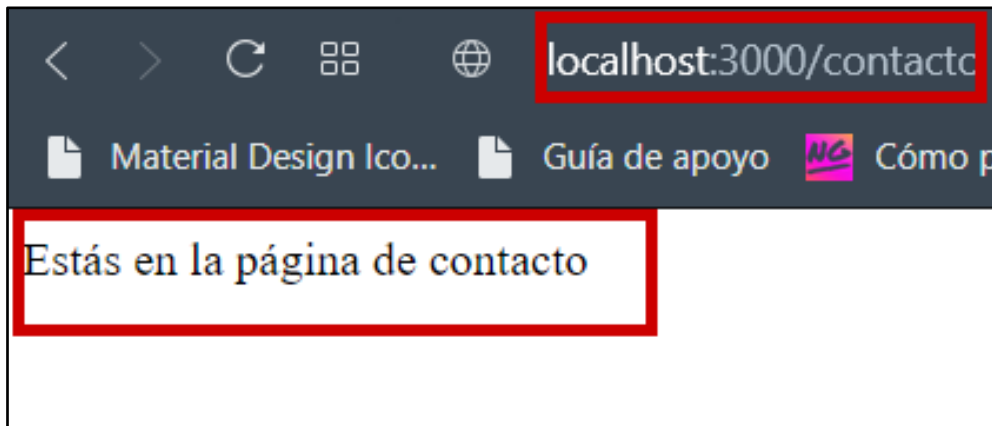
JS app.js

```

4   const port = 3000;
5
6   app.use(express.static(__dirname + "/public"))
7
8   app.get("/", (req, res) => {
9     res.send("Mi respuesta desde express");
10  });
11
12  app.get("/contacto", (req, res) => {
13    res.send("Estás en la página de contacto");
14  });
15
16  app.use((req, res, next) => {
17    res.status(404).sendFile(__dirname + "/public/404.html")
18  })
19
20
21  app.listen(port, () => {
22    console.log('Servidor preparado en el puerto', port);
23  });

```

Guardamos, y visualizamos en nuestro navegador, actualizamos la página de contacto, y efectivamente devuelve el **string** que teníamos definido.

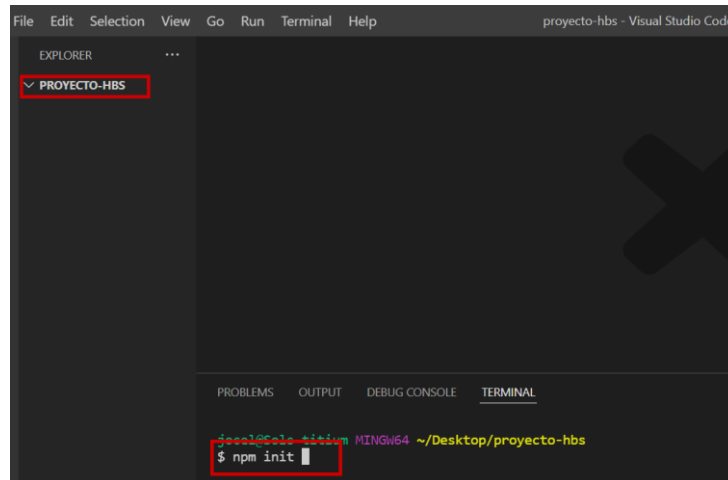


Con lo que hemos visto y estudiado, ya sabemos configurar nuestro primer servidor con Express, y también aprendimos sobre las páginas estáticas para conocer el funcionamiento de las rutas con Express.

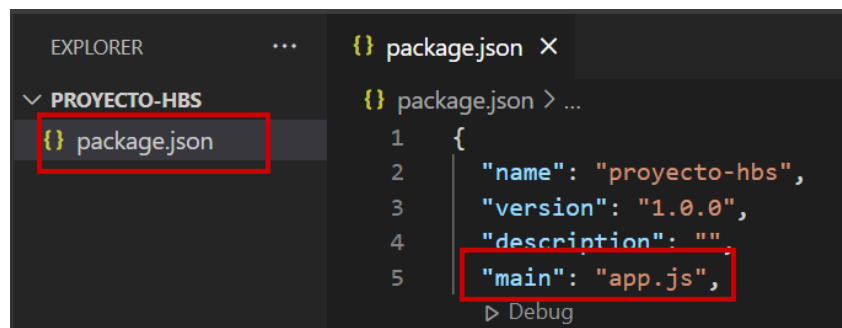
#### **EXERCISE 4: UTILIZANDO EXPRESS CON MOTOR DE PLANTILLA HANDLEBARS Y BOOTSTRAP.**

En el siguiente ejercicio aprenderemos a utilizar el sistema de plantillas **Handlebars** junto a Express. Lo primero que haremos será crear una carpeta, en la cual tendremos nuestro proyecto, ésta llevará por nombre “proyecto-hbs”, la abrimos en Visual Studio Code, y en la terminal escribimos **npm init -y** para poder trabajar con **node**.

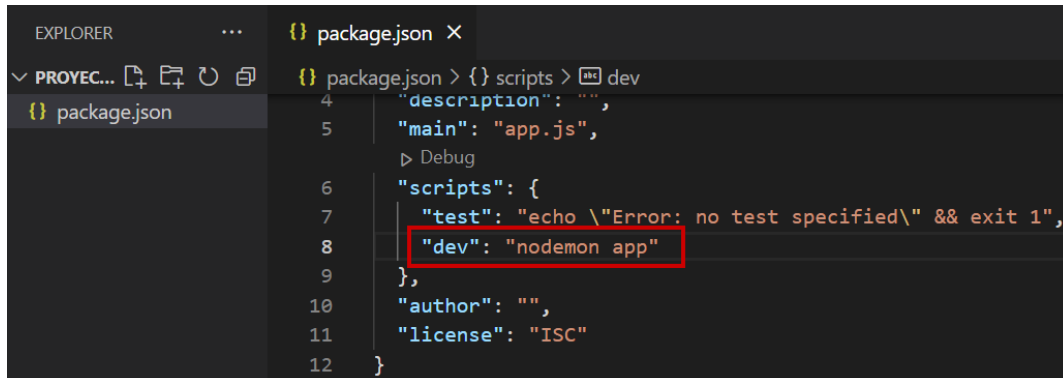
Se agrega **-y** para que todas las preguntas que hace la instalación de **node** sean “yes”, y así no nos demorarnos en responder cada una.



Una vez instalado **node**, abriremos el archivo **package.json**, modificaremos el **main**, y cambiaremos **index.js** por **app.js**, que será el archivo que utilizaremos para el servidor y las rutas, tal como lo hicimos en el ejercicio anterior.

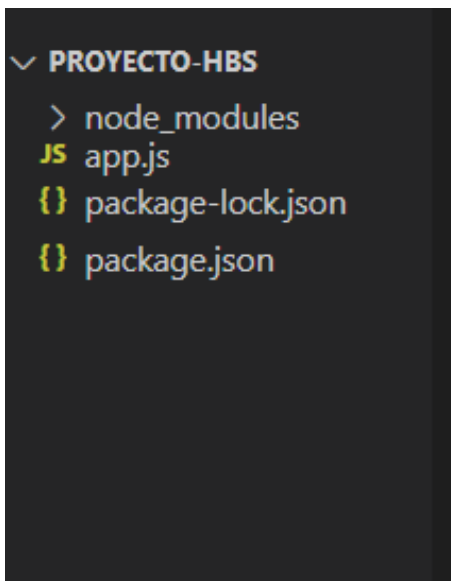


En el mismo archivo **package.json** modificaremos el script, y agregaremos **dev: nodemon app**, para poder trabajar con la herramienta **nodemon**. Como en el ejercicio anterior lo instalamos globalmente, en éste no será necesario volver a instalarlo, y "app" se refiere al archivo que crearemos para el servidor.



```
EXPLORER  ...  {} package.json X
PROYEC...  {} package.json
package.json
4  "description": "",
5  "main": "app.js",
   Debug
6  "scripts": {
7    "test": "echo \"Error: no test specified\" && exit 1",
8    "dev": "nodemon app"
9  },
10 "author": "",
11 "license": "ISC"
12 }
```

Ahora crearemos el archivo `app.js`, y aquí tendremos el servidor que nos permitía utilizar `express`, tal como lo hicimos en el primer ejercicio.

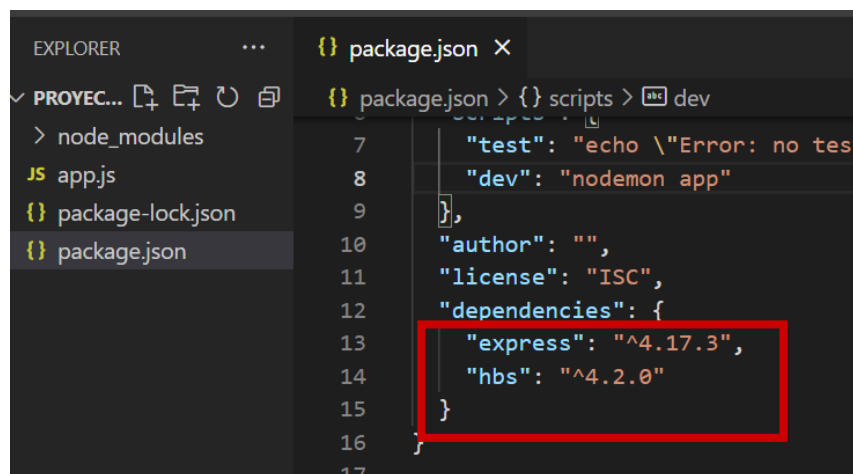


```
PROYECTO-HBS
> node_modules
JS app.js
{} package-lock.json
{} package.json
```

Volvemos a la terminal, e instalaremos `express` y `hbs` con el comando `npm i express hbs`. Recordemos que se puede instalar más de un paquete en una línea de comando.

```
js@Cale titium MTN64 ~/Desktop/proyecto-hbs
$ npm i express hbs
```

Para comprobar que se instalaron correctamente **express** y **hbs**, nos dirigimos al archivo **package.json**, y revisaremos las dependencias, tal como se ven en la siguiente imagen.



```
{
  "name": "proyecto-hbs",
  "version": "1.0.0",
  "scripts": {
    "test": "echo \"Error: no test specified\"",
    "dev": "nodemon app.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.3",
    "hbs": "^4.2.0"
  }
}
```

Lo siguiente que haremos será empezar con la configuración comenzaremos con las configuraciones en el archivo **app.js**, lo primero que tendremos será la instancia de **express** para configurar el servidor

```
JS app.js
JS app.js > ...
1 const express = require('express')
2 const app = express();
3 const port = 3000;
4
```

Lo siguiente que tendremos será lo necesario para utilizar el motor de plantilla Handlebars.

Analizaremos el código por partes, y lo primero que vemos es que se debe requerir a **hbs**.



```
//hbs  
const hbs = require('hbs');
```

Lo segundo, es el registro de los **partials**; éstos son trozos de códigos que se pueden reutilizar, y los explicaremos a medida que el ejercicio avance.

```
hbs.registerPartials(__dirname + '/views/partials', function(err) {});
```

Tercero, se debe declarar el motor de plantillas que utilizaremos, en este caso será **hbs**.

```
app.set('view engine', 'hbs');
```

Cuarto, indicaremos que en la carpeta **views** pondremos todos los archivos estáticos, para visualizar las distintas páginas que tendrá nuestra web.

```
app.set ("views", __dirname + "/views");
```

Quinto, tendremos, tal como lo hicimos en el ejercicio anterior, la carpeta **public** que nos servirá para nuestros archivos JavaScript, CSS, imágenes, entre otros.





```
//Archivos estáticos  
app.use(express.static(__dirname + "/public"))
```

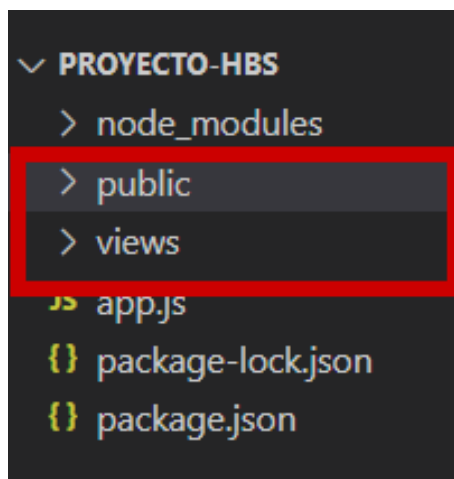
Sexto, tendremos el espacio para crear las rutas que utilizaremos, las cuales iremos creando a medida que el ejercicio avance y, por último, tendremos el código donde nuestro servidor “escucha”, y la salida del puerto que utilizaremos.

```
//Rutas  
  
//Iniciar servidor  
app.listen(port, () => {  
  console.log('Servidor preparado en el puerto', port)  
})
```

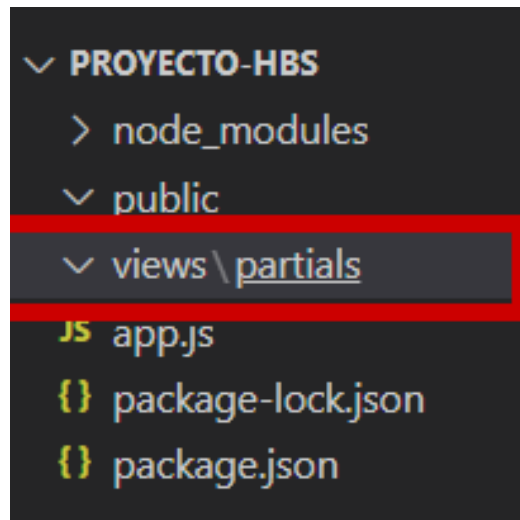
El código completo quedará de la siguiente manera.

```
JS app.js  X
JS app.js > ...
1  const express = require('express')
2  const app = express();
3  const port = 3000;
4
5  //hbs
6  const hbs = require('hbs');
7  hbs.registerPartials(__dirname + '/views/partials', function (err) { });
8  app.set('view engine', 'hbs');
9  app.set("views", __dirname + "/views");
10
11 //Archivos estáticos
12 app.use(express.static(__dirname + "/public"))
13
14 //Rutas
15
16
17
18 //Iniciar servidor
19 app.listen(port, () => {
20   console.log('Servidor preparado en el puerto', port)
21 })
22 }
```

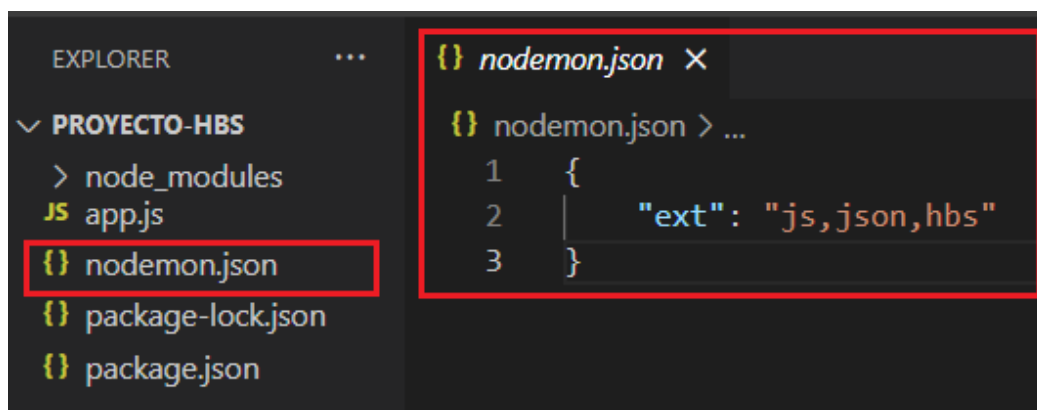
Ahora, necesitamos crear las carpetas que utilizaremos, en este caso, **public** y **views** en la raíz de nuestro proyecto.



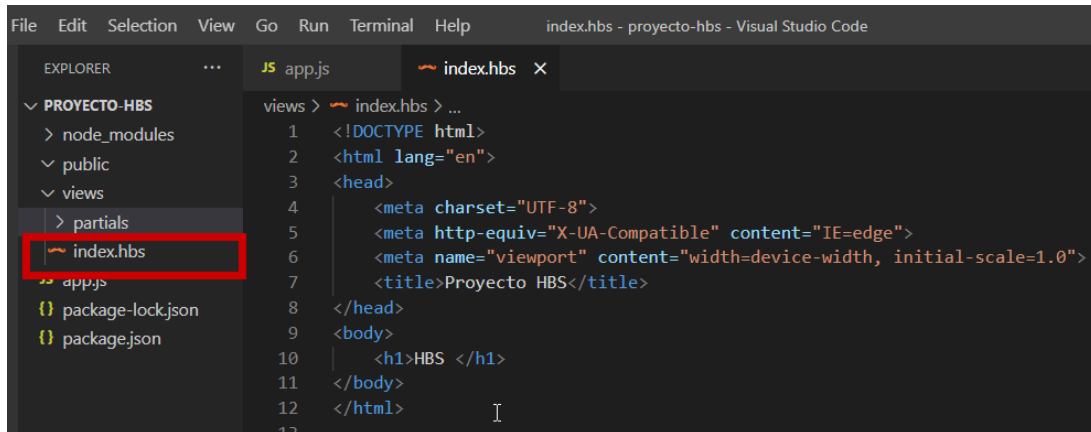
Y dentro de la carpeta **views**, crearemos la carpeta **partials**.



Ahora, lo que nos queda hacer es crear en la carpeta raíz, un archivo que se llamará **nodemon.json**. Esto es para que **nodemon** sea capaz de leer nuestros archivos que estarán en la carpeta **partials**, y así detectará los cambios que hagamos en los archivos con extensión **hbs**, sin necesidad de reiniciar el servidor cada vez que suceda uno. Dentro del archivo **nodemon.js**, colocaremos la información que veremos en la imagen a continuación.



Ahora, crearemos nuestro primer archivo, y lo haremos dentro de la carpeta **Views**, éste se llamará **index.hbs**, y tendrá la estructura básica de un archivo **html** con el título **Proyecto HBS**, y un h1 que dirá **HBS**.



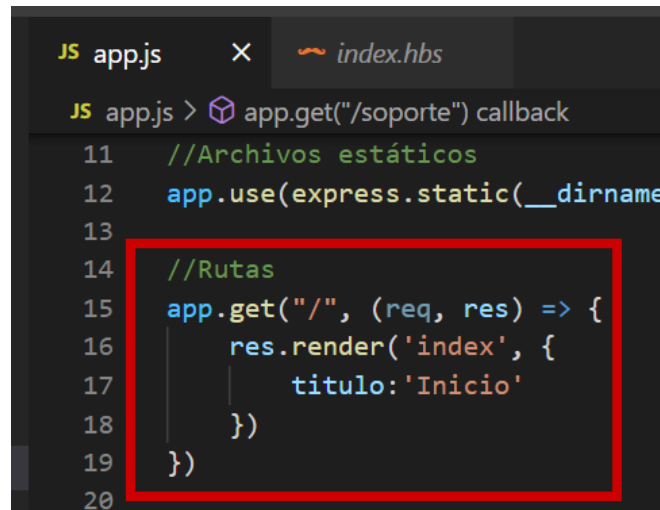
```

File Edit Selection View Go Run Terminal Help index.hbs - proyecto-hbs - Visual Studio Code

EXPLORER
PROYECTO-HBS
  > node_modules
  > public
  > views
    > partials
    index.hbs
  app.js
  package-lock.json
  package.json

views > index.hbs > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Proyecto HBS</title>
8 </head>
9 <body>
10   <h1>HBS </h1>
11 </body>
12 </html>
13
  
```

Al crear una vista, también tenemos que “pintarla”. Para ello, debemos generar la ruta en el archivo `app.js`, empezando por `app.get` para requerirla, y entre paréntesis tendremos en comillas la ruta que en este caso será la ruta raíz `/`; y a continuación, un requerimiento y una respuesta, que dentro de su función renderizará el `index` (no es necesario colocar la extensión `hbs`); por último, tendremos un dato que se llamará título, generalmente los datos que vendrán son de algún servidor, y en este caso, pueden venir en un objeto.



```

JS app.js  X  index.hbs

JS app.js > app.get("/soporte") callback
11 //Archivos estáticos
12 app.use(express.static(__dirname
13
14 //Rutas
15 app.get("/", (req, res) => {
16   res.render('index', {
17     titulo: 'Inicio'
18   })
19 })
20
  
```

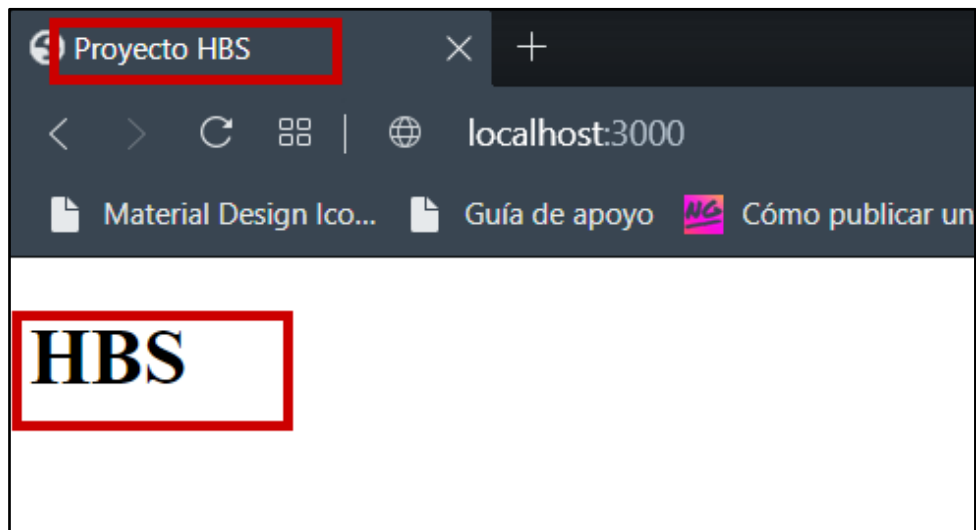
Guardamos todos los cambios, nos dirigimos a la consola de VSC, y escribiremos el comando `npm run dev`; al presionar enter podremos ver el mensaje de servidor, preparado junto al número del puerto que declaramos.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

> nodemon app

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node app.js`
Servidor preparado en el puerto 3000
```

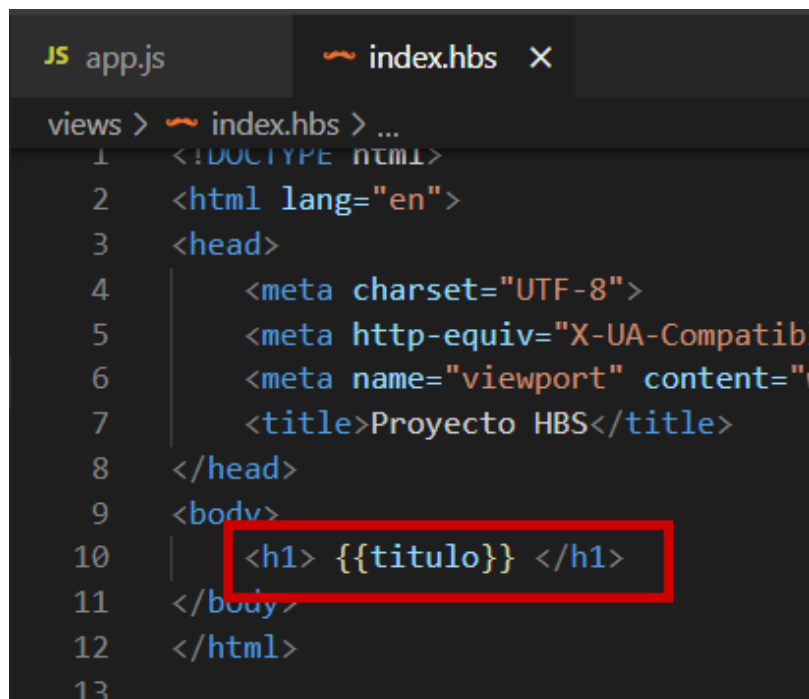
Si revisamos el navegador, podremos ver el título que definimos, y el `h1` que teníamos en nuestro archivo `index.hbs`.



De esta manera notaremos que se está leyendo nuestro archivo. Ahora, necesitamos leer el título de la ruta raíz desde el archivo `app.js`, y que éste se vea en la página de inicio que es `index.hbs`.

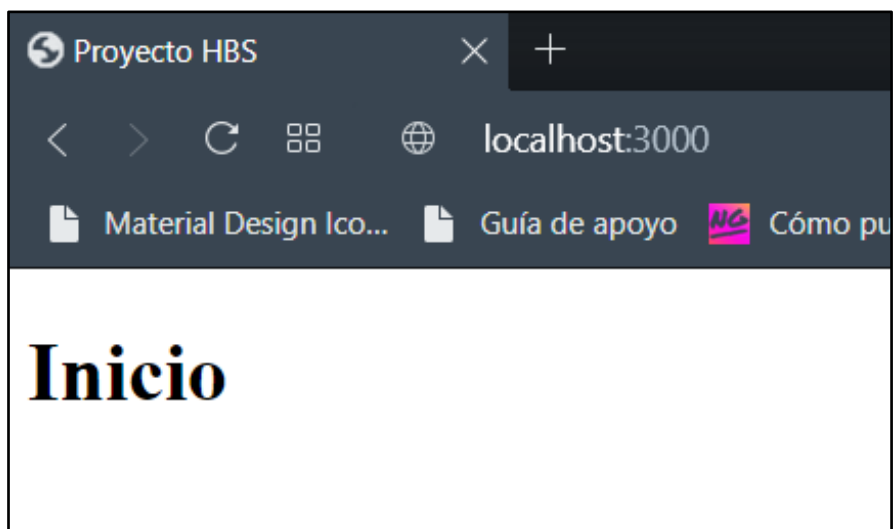
```
//Rutas
app.get("/", (req, res) => {
  res.render('index', {
    titulo: 'Inicio'
  })
})
```

Nos dirigiremos al archivo `index.hbs`, y en el `h1` agregaremos entre doble llaves `"titulo"`, que hace referencia al título del archivo `app.js`.

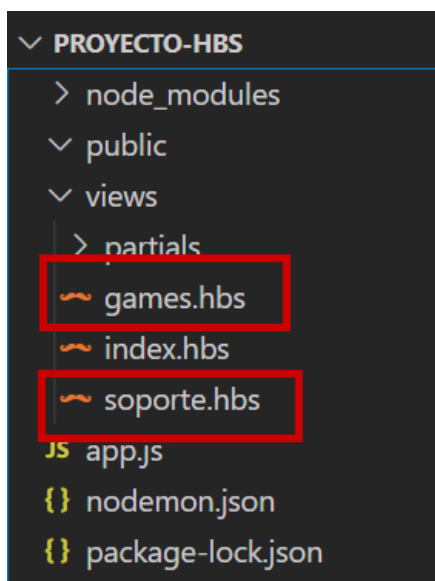


```
JS app.js  index.hbs X
views > index.hbs > ...
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatib
6   <meta name="viewport" content="v
7   <title>Proyecto HBS</title>
8 </head>
9 <body>
10  <h1> {{titulo}} </h1>
11 </body>
12 </html>
13
```

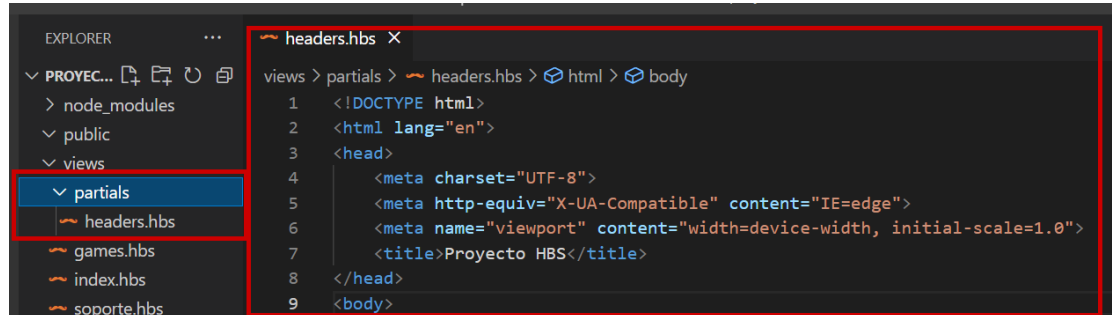
Si revisamos el navegador, podemos ver que efectivamente trae el título "Inicio" que habíamos definido en `app.js`.



Ahora, crearemos dos nuevas vistas: una se llamará `soporte.hbs`, y la otra `games.hbs`.



Generalmente, en estas vistas se crearía la estructura base de un `html`, pero si lo hacemos así, estaríamos repitiendo código y no es lo más recomendable. Entonces, para que no suceda, utilizaremos los `partials`. Éstos nos sirven para no tener que duplicar nuestro código, y que éste también sea dinámico; el primero que crearemos será `headers.hbs`, y estará dentro de la carpeta `partials`, y a su vez, dentro de la carpeta `views` donde sólo tendremos la cabecera que tendrán nuestras vistas.



EXPLORER

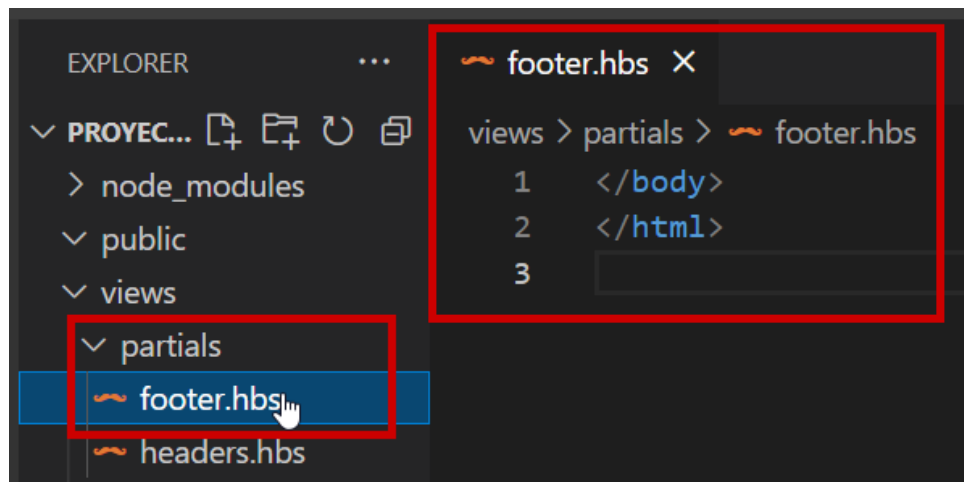
- PROYEC...
- node\_modules
- public
- views
  - partials
    - headers.hbs
    - games.hbs
    - index.hbs
    - soporte.hbs

headers.hbs

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" content="IE=edge">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Proyecto HBS</title>
8 </head>
9 <body>
  
```

También tendremos nuestro archivo **footer.hbs** en la carpeta **partials**.



EXPLORER

- PROYEC...
- node\_modules
- public
- views
  - partials
    - footer.hbs
    - headers.hbs

footer.hbs

```

1 </body>
2 </html>
3
  
```

Ahora, lo que necesitamos es que el header y el footer se observen en nuestra vista principal **index.hbs**. Para ello, utilizaremos la nomenclatura **{{>}}**, y dentro tendrá el nombre del archivo **partials** que queramos utilizar.



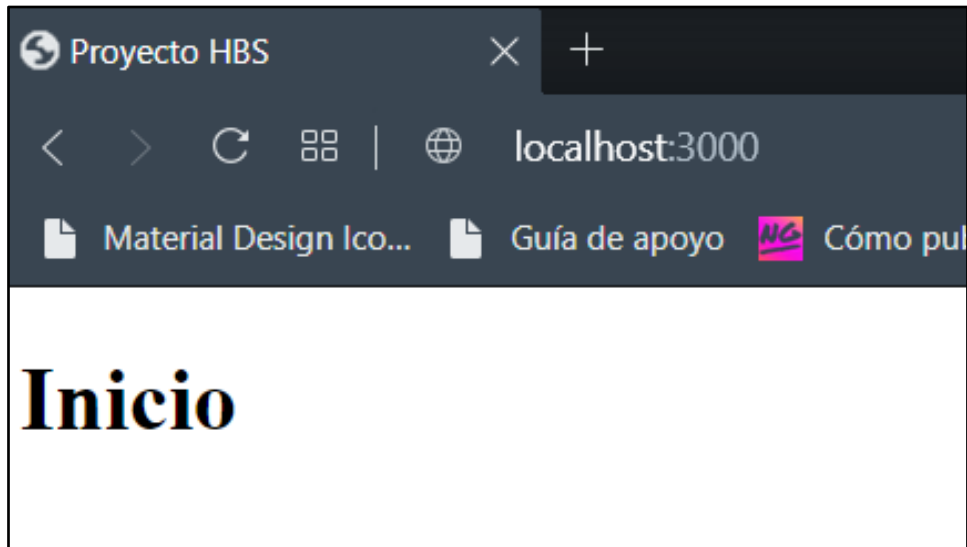
views > index.hbs > ...

```

1 {{> header}}
2
3   <h1> {{titulo}} </h1>
4
5 {{> footer}}
6
7
  
```



Si guardamos, y nos dirigimos a nuestro navegador, podremos observar que todo funciona bien, se ve el **title** que definimos, y el título inicio sin ningún problema.




Ahora, probaremos hacer el título de nuestro **header** dinámico. Para esto, nos vamos al **partials header**, y pondremos entre doble llaves **title**.

```

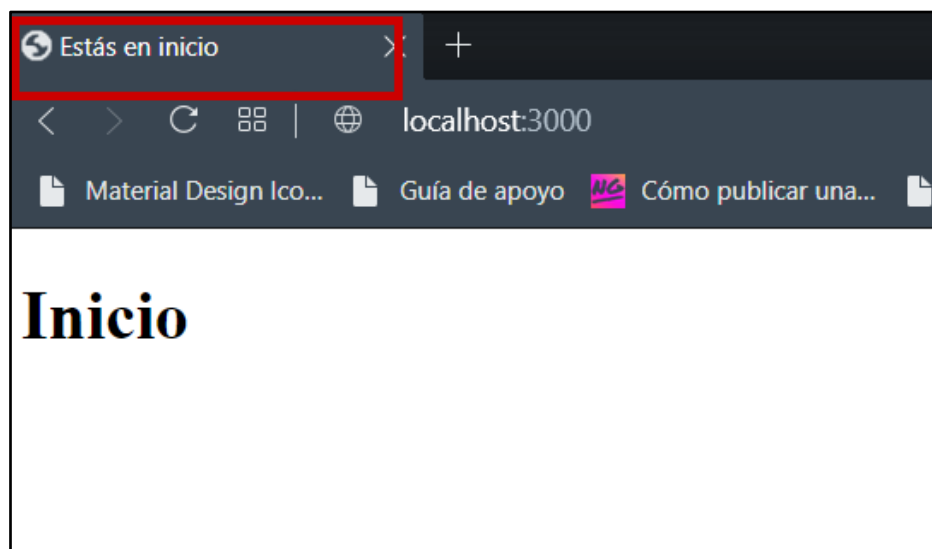
headers.hbs × index.hbs
views > partials > headers.hbs > html > body
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta http-equiv="X-UA-Compatible" >
6   <meta name="viewport" content="width
7   <title> {{title}} </title>
8 </head>
9 <body>
  
```

Para poder leer ese `title` desde nuestro archivo `index.hbs`, debemos colocar `title = "Estás en inicio"`.

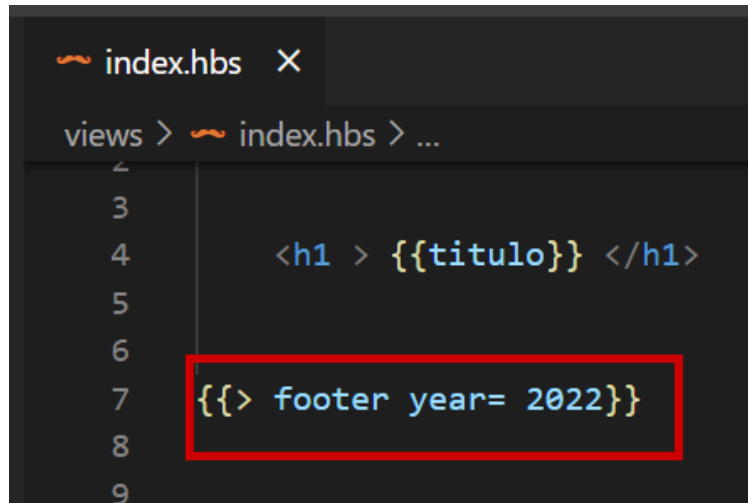


```
index.hbs X
views > index.hbs > ...
1  {{> headers title='Estás en inicio' }}
2
3  <h1> {{titulo}} </h1>
4
5  {{> footer}}
```

Guardamos, y recargamos nuestro navegador. Podremos ver que el `title` de nuestra página cambió.

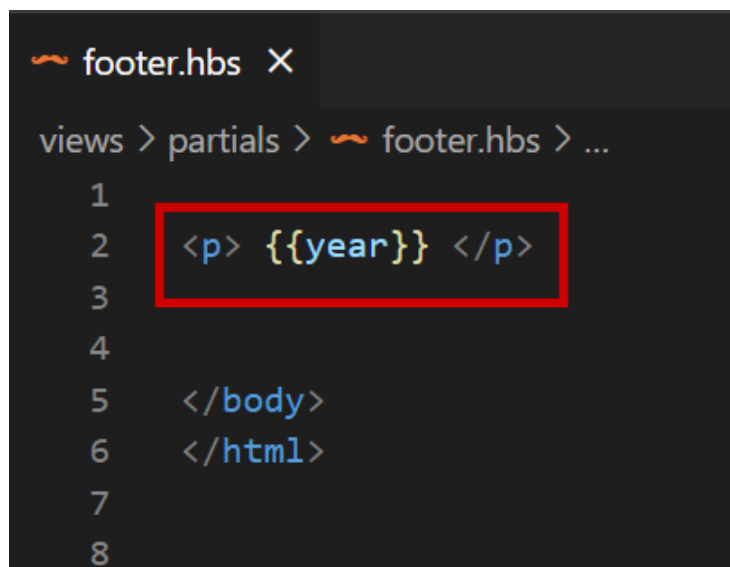


También dejaremos nuestro **footer** dinámico. Para esto, en nuestro archivo **index.hbs** agregaremos **year = 2022**, y guardamos.



```
index.hbs X
views > index.hbs > ...
1
2
3
4     <h1> {{titulo}} </h1>
5
6
7     {{> footer year= 2022}}
8
9
```

Y en nuestro archivo **footer.hbs**, colocaremos **year**.



```
footer.hbs X
views > partials > footer.hbs > ...
1
2     <p> {{year}} </p>
3
4
5 </body>
6 </html>
7
8
```

Guardamos, y actualizamos nuestro navegador. Veremos el footer 2022.



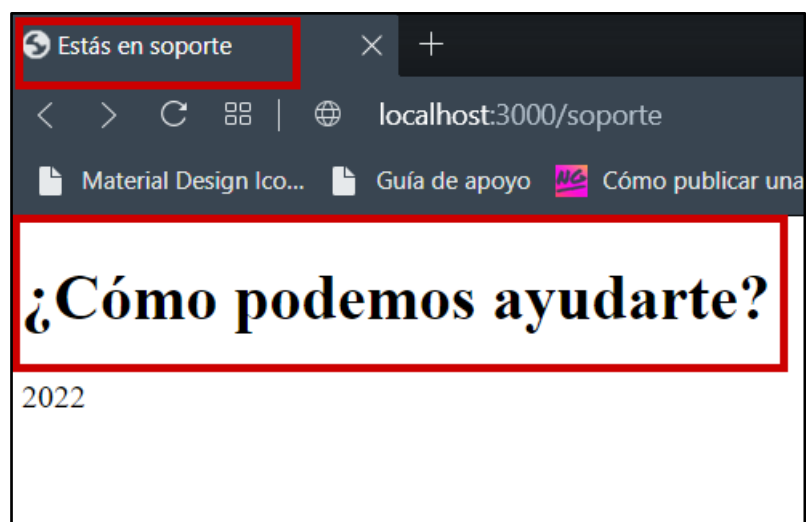
Ahora, empezaremos a agregar la ruta soporte a nuestro archivo `app.js`, y le pondremos: un título, un estado, y un tipo de soporte `"Recuperación de la cuenta"`.

```
app.get("/soporte", (req, res) => {  
  res.render('soporte', {  
    titulo: '¿Cómo podemos ayudarte?',  
    estado: true,  
    soporte: 'Recuperación de la cuenta'  
  })  
})
```

En nuestra vista `soporte.hbs` agregaremos lo necesario para que se vean los datos requeridos, copiamos el contenido del archivo `index.hbs`, lo pegamos en `soporte.hbs`, y cambiaremos el título por `“Estás en soporte”`.

```
soporte.hbs X
views > soporte.hbs > ...
1  {{> header title="Estás en soporte"}}
2
3
4  <h1 > {{titulo}} </h1>
5
6
7  {{> footer year= 2022}}
8
```

Guardamos, actualizamos el navegador colocando `localhost:3000/soporte`, y veremos todo lo correspondiente a la vista soporte.



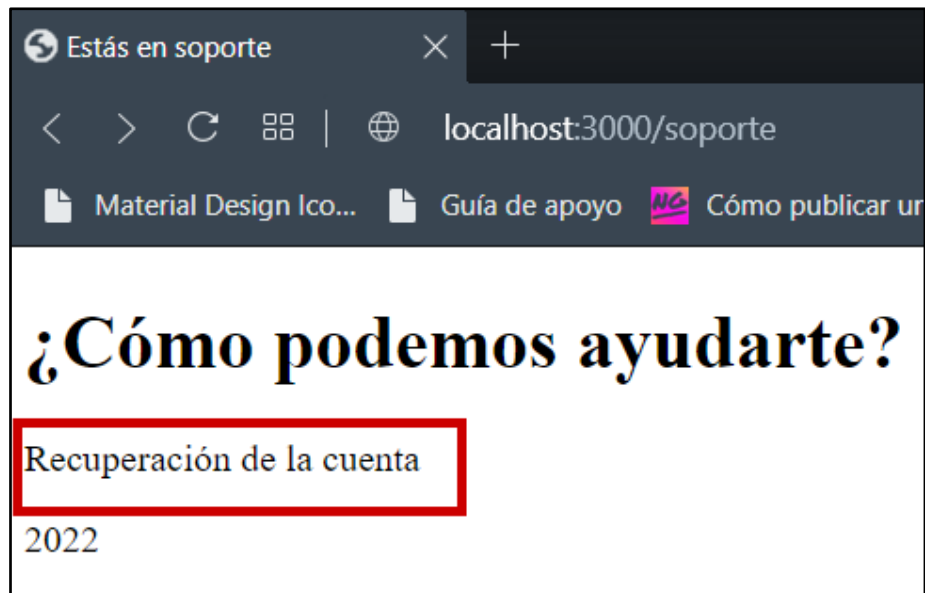
Ahora, utilizaremos el condicional **if** para poder evaluar el estado que se definió en la ruta soporte del archivo **app.js**.

```
soporte.hbs X
views > soporte.hbs > ...
1  {{> headers title="Estás en soporte"}}
2
3  <h1 > {{titulo}} </h1>
4
5      {{#if estado}}
6
7
8      {{/if}}
9
10 {{> footer year= 2022}}
11
```

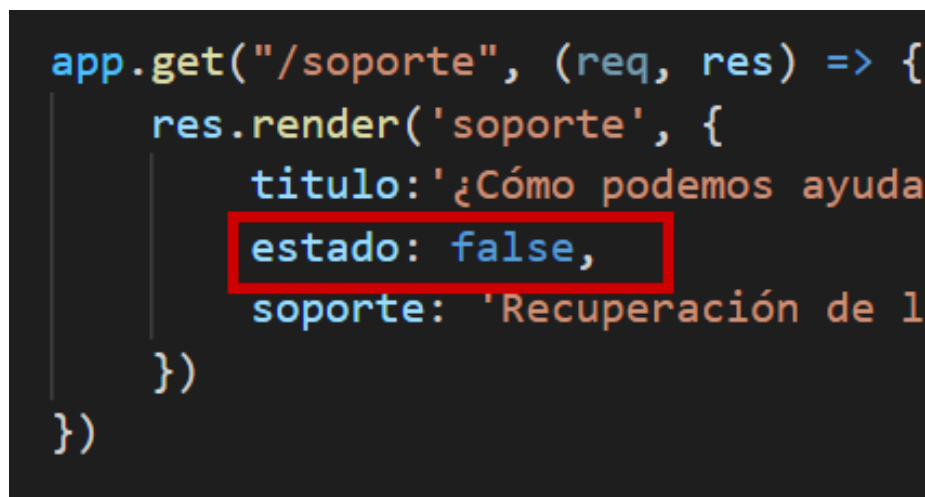
Todo lo que esté dentro del **if** se verá sólo si el estado es **true**, entonces colocaremos un párrafo con el soporte que queremos mostrar.

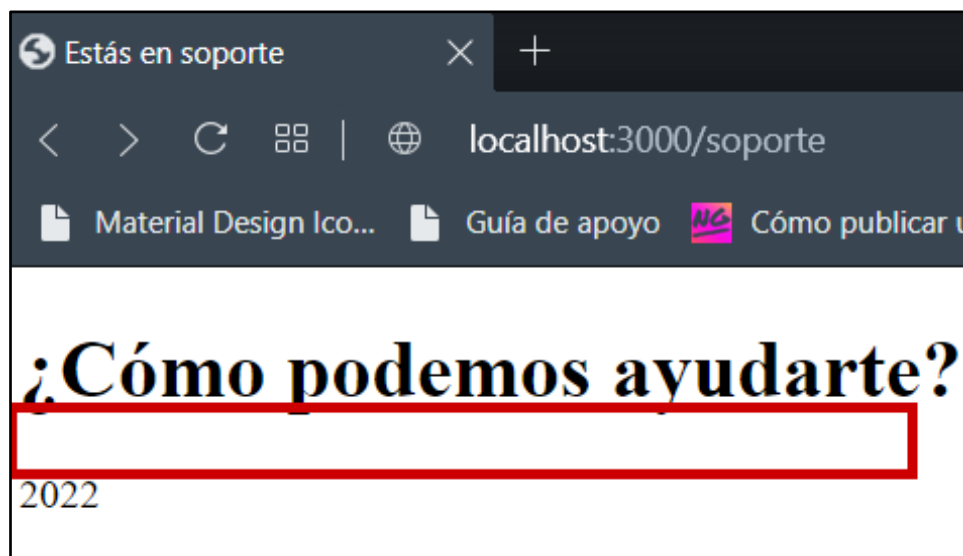
```
soporte.hbs X
views > soporte.hbs > ...
1  {{> headers title="Estás en soporte"}}
2
3  <h1 > {{titulo}} </h1>
4
5      {{#if estado}}
6
7      <p> {{soporte}} </p>
8
9      {{/if}}
10
11 {{> footer year= 2022}}
12
```

Guardamos todo, y actualizamos el navegador para verificar que nos muestra el soporte que invocamos.



Ahora, probaremos colocando el estado del soporte como falso, y al actualizar el navegador, podemos ver que no se muestra el soporte que teníamos definido.





Si quisiéramos “pintar” algo cuando estado no exista, tenemos la sentencia **else**, la cual nos permitirá colocar un párrafo que dirá: “Lo sentimos no podemos ayudarte”.

```
soporte.hbs X
views > soporte.hbs > ...
1  {{> headers title="Estás en soporte"}}
2
3  <h1 > {{titulo}} </h1>
4
5      {{#if estado}}
6
7      <p> {{soporte}} </p>
8
9      {{else}}
10     <p>Lo sentimos no podemos ayudarte</p>
11
12     {{/if}}
13
14  {{> footer year= 2022}}
```



Ahora, conoceremos `each`, y veremos dos ejemplos para poder utilizarlo. Lo primero que haremos será crear la ruta para “games”, en el que tendremos un título y un `array` con dos juegos.

```
app.get("/games", (req, res) => {  
  res.render('games', {  
    titulo: 'Juegos',  
    games: ['Valorant', 'Heroes of the Storm']  
  })  
})
```

Lo segundo, será pintar esos juegos en nuestra vista “games”. Para esto, utilizaremos las etiquetas `ul`, y dentro estará `each` junto al nombre de nuestro Array `“games”`. Esto nos permitirá recorrer el arreglo.

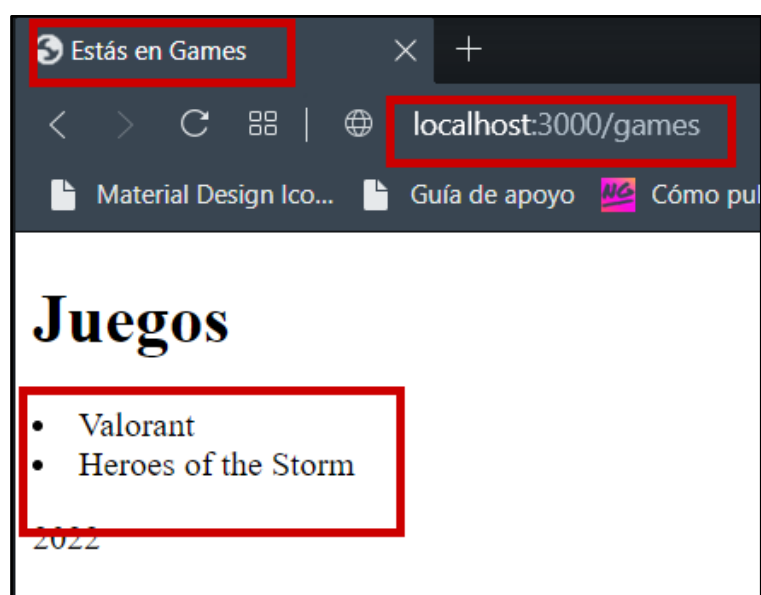
```
games.hbs X  
views > games.hbs > ...  
1  {{> headers title="Estás en Games"}}  
2  
3  <h1 > {{titulo}} </h1>  
4  
5  {{#each games}}  
6  
7  
8  {{/each}}  
9  
10 {{> footer year= 2022}}  
11  
12
```

Dentro de nuestro `each` colocaremos una etiqueta `li`, y dentro, con llaves dobles, `this`, que se comporta como los elementos que tenemos en el arreglo.



```
games.hbs X
views > games.hbs > ...
1  {{> headers title="Estás en Games"}}
2
3  <h1> {{titulo}} </h1>
4
5  {{#each games}}
6    <li> {{this}}</li>
7
8  {{/each}}
9
10
11 {{> footer year= 2022}}
```

Revisamos la URL `localhost:3000/games` en nuestro navegador, y podemos ver efectivamente los dos juegos guardados en el arreglo que hicimos.



Pero, ¿qué pasaría si queremos agregar un Array de objetos? A continuación, iremos al archivo `app.js`, a la ruta de games, y modificaremos el título poniendo `"lista de juegos"`; además, agregaremos un arreglo de objetos que tendrá: id, nombre, y género del juego.

```
app.get("/games", (req, res) => {  
  res.render('games', {  
    titulo: 'Lista de juegos',  
    games: [{ id: '1', nombre: 'Valorant', genero: 'Shooter'},  
             { id: '2', nombre: 'Heroes of the storm', genero: 'Moba'},  
             { id: '3', nombre: 'League of legends', genero: 'Moba'},  
             { id: '4', nombre: 'Overwatch', genero: 'Acción por equipos'}  
          ]  
    })  
  })  
})
```

Antes de continuar con nuestro código, y ya que tenemos la base del motor de plantillas `Handlebars`, haremos algunas modificaciones a nuestras vistas para mostrar la información que queremos, pero de una manera más amigable.

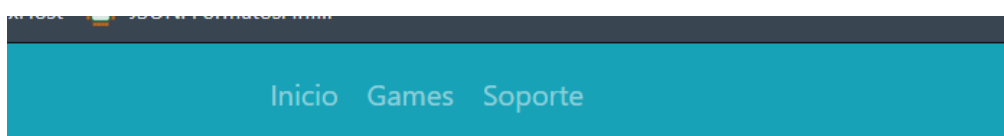
Para eso utilizaremos Bootstrap y agregaremos el `CDN` en el archivo `headers.hbs`. En este mismo archivo, crearemos un menú básico, donde se enlazarán las vistas: Inicio, Games, y Soporte.

```

1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta http-equiv="X-UA-Compatible" content="IE=edge">
7   <meta name="viewport" content="width=device-width, initial-
8 scale=1.0">
9   <title> {{title}} </title>
10
11   <link rel="stylesheet"
12 href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/boo
13 tstrap.min.css"
14   integrity="sha384-
15 JcKb8q3iqJ61gNV9KGb8thSsNjpSL0n8PARn9HuZOnIxN0hoP+VmmDGMN5t9UJ0Z
16 " crossorigin="anonymous">
17 </head>
18
19 <body>
20   <nav class="navbar navbar-expand-sm bg-info navbar-dark
21 justify-content-center">
22     <ul class="navbar-nav">
23       <li class="nav-item">
24         <a class="nav-link" href="/">Inicio</a>
25       </li>
26       <li class="nav-item">
27         <a class="nav-link" href="/games">Games</a>
28       </li>
29       <li class="nav-item">
30         <a class="nav-link" href="/soporte">Soporte</a>
31       </li>
32     </ul>
33   </nav>

```

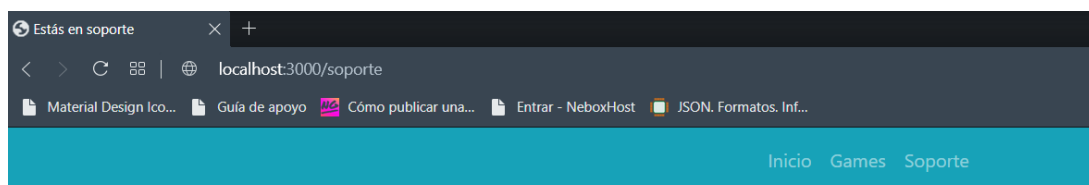
Si vamos al navegador, veremos el menú creado.



A nuestra vista soporte le agregaremos la etiqueta `<h1>` del título la clase `p-5`, para incluir un espacio entre el título y el menú; además, un componente de **Bootstrap** llamado **jumbotron**, para mostrar contenido de forma destacada, en este caso, la información del **if**.

```
1 {{> headers title="Estás en soporte"}}
2
3 <h1 class="p-5">{{ titulo }} </h1>
4
5 <div class="jumbotron jumbotron-fluid" style="background-
6 color:#8C9EFF">
7   <div class="container">
8
9     {{#if estado}}
10
11     <p style="font-size: 25px;">{{soporte}}</p>
12
13     {{else}}
14     <p>Lo sentimos no podemos ayudarte</p>
15
16     {{/if}}
17   </div>
18 </div>
19
20 {{> footer year= 2022}}
```

Si nos dirigimos al navegador, se verá de la siguiente manera.



## ¿Cómo podemos ayudarte?

Recuperación de la cuenta

A nuestra vista footer le agregaremos en la etiqueta `<p>`, la clase `text-center` y la palabra que diga Footer.

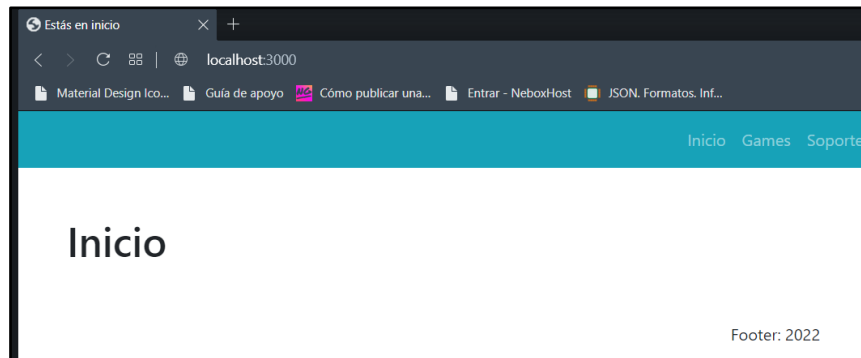
```
footer.hbs X
views > partials > footer.hbs > ...
1
2 <p class="text-center" > Footer: {{year}} </p>
3
4
5 </body>
6 </html>
7
```

Y se verá desde el navegador de la siguiente manera.



Y en la vista `index.hbs`, solo le agregaremos una `clase p-5` a la etiqueta `h1`, para ver el título separado del menú.

```
index.hbs X
views > index.hbs > ...
1 {{> headers title="Estás en inicio"}}
2
3 <h1 class="p-5" > {{titulo}} </h1>
4
5 {{> footer year= 2022}}
6
```



Ahora, a la vista games le agregaremos una tabla de Bootstrap, que muestre todos los juegos que tenemos en el arreglo `games`, junto al: id, nombre, y género del juego.

Para eso utilizamos nuevamente `#each`, que recorrerá el arreglo, y a `this`, que representaba cada uno de los datos de nuestro arreglo, y le agregamos la propiedad que queremos mostrar del objeto, tal como se muestra a continuación.

```

1  {{> headers title="Estás en Games"}}
2  <h1 class="text-center p-5"> {{ titulo }} </h1>
3
4  <div class="container p-5">
5      <table class="table table-dark table-hover">
6          <thead class="text-center">
7              <tr >
8                  <th scope="col">Id</th>
9                  <th scope="col">Nombre Juego</th>
10                 <th scope="col">Género del Juego</th>
11             </tr>
12         </thead>
13
14         <tbody class="text-center">
15             <tr>
16                 {{#each games}}
17
18                 <th scope="row">{{this.id}} </th>
19                 <td>{{this.nombre}}</td>
20                 <td>{{this.genero}}</td>
21             </tr>
22
23             {{/each}}
24
25         </tbody>
26     </table>
27 </div>
28
  
```

```
29 {{>footer year= 2022}}
```

Y finalmente, veremos en el navegador todos los juegos que teníamos guardados en Games.



## Lista de juegos

Id	Nombre Juego	Género del Juego
1	Valorant	Shooter
2	Heroes of the storm	Moba
3	League of legends	Moba
4	Overwatch	Acción por equipos

Footer: 2022

De esta manera, aprendimos a utilizar **Express** con el motor de plantilla **Handlebars** y **Bootstrap**.