NODE JS

Diego Caldera Beltrán - Pablo González López

ÍNDICE

BLOQUE 1: Introducción e instalación

1- ¿Qué es NodeJS?

2- Utilidades de NodeJS.

3- Ventajas y desventajas de NodeJS.

4- Comparación con otros lenguajes.

5- Primeros pasos con NodeJS.

¿Qué es Node.js?

Es un entorno de ejecución de código abierto y multiplataforma que permite ejecutar JavaScript fuera del navegador, haciendo uso de un servidor. Es una herramienta potente para crear aplicaciones web de gran escalabilidad y alto rendimiento. Esto permite:

- Usar JavaScript tanto en el backend como en el frontend.
- Ofrecer operaciones de entrada y de salida, por ejemplo en chats en tiempo real, sistemas de Streaming o APIs.

Utilidades de NodeJS

Node.js destaca en una serie de escenarios concretos donde sus características (como la asincronía, la ligereza y la escalabilidad) lo convierten en una solución eficiente.

A continuación, exploramos algunos de los contextos en los que Node.js se utiliza con mayor frecuencia:

- **Microservicios**
- **APIs REST**
- **Aplicaciones con alta concurrencia**
- **Backends para aplicaciones móviles**

Ventajas y desventajas de NodeJS

Ventajas:

- **Escalabilidad**: Gestiona gran cantidad de solicitudes, sin bloquear el proceso principal, por lo que se puede ir adaptando según las necesidades.

- **Comunidad**: Gracias a que es de código abierto muchos profesionales actualizan el software a diario, los recursos son siempre actualizados.

- **Gran Ecosistema**: Con NPM se puede reutilizar y compartir código fácilmente y hay muchos frameworks que puedes usar con Node.js.

Desventajas:

- **Asincronía**: Puede resultar una desventaja para aquellos usuarios que están acostumbrados a una programación síncrona.

- **Herramienta relativamente nueva**:
Esto provoca que haya cambios frecuentes en la API, en ocasiones faltan funcionalidades y características que generan inestabilidad en los desarrollos.

Otros lenguajes...

A continuación, podemos ver las comparaciones con otros lenguajes y sus características:

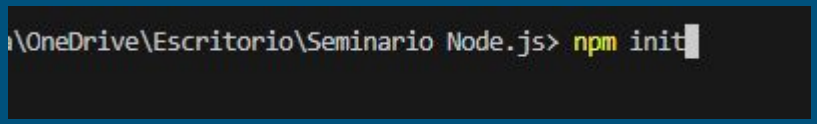
CARACTERÍSTICAS	node	php	.NET
Lenguaje Principal	JavaScript	PHP	C#
Modelo Ejecución	Asíncrono/Eventos	Síncrono	Multi-Hilo
Velocidad	Muy rápida	Rápida Web	Alta
Escalabilidad	Muy Alta.	Media	Muy alta

Primeros pasos con NodeJS

-Abrimos VS Code.

-Cada uno crea su carpeta en el WorkSpace que destinará al seminario.

-Ejecutamos en el WorkSpace el siguiente comando:



-El proyecto se creará y se ejecutará con el comando “node + *nombreArchivo.js*”. En nuestro caso, será con “node index.js”.

-Package.json

BLOQUE 2: Conceptos fundamentales y asincronía

1- Event Loop y cómo manejar múltiples solicitudes.

2- Callbacks, Promesas, async/await.

3- Introducción al uso de módulos.

Event Loop y cómo manejar múltiples solicitudes

El event loop es el que se encarga de implementar las operaciones asíncronas. Node no crea un hilo por cada petición (como otros lenguajes), sino que usa un único hilo que deja las tareas lentas al sistema y sigue ejecutando otras cosas mientras tanto.

La asincronía provoca que se ejecuten varias tareas en paralelo o sin esperar a que acabe lo anterior.

Seminario Node.js > JS index.js > ...

```
1  console.log("Hola mundo");
2
3  setTimeout(() => {
4    console.log("Hola mundo despues de 3 segundos.");
5  }, 3000);
6
```

Consola:

```
PS C:\Users\Luis Caldera\OneDrive\Escritorio\Seminario Node.js> node index.js
Hola mundo
Hola mundo despues de 3 segundos.
```

Callbacks, Promesas y async/await

Un Callback es una función que se pasa como argumento a otra y se ejecuta después de que una tarea asíncrona termine.

JS callbacks.js > ...

```
1   function notificar() {
2     console.log("Tarea completada");
3   }
4
5   setTimeout(notificar, 1000);
```

Cuando una operación depende de otra, el código se anida.

```
PS C:\Users\pabli\Desktop\SeminarioNode> node callbacks.js
Tarea completada
```

Callbacks, Promesas y async/await

Para contrarrestar los problemas de anidamiento que conllevan los callbacks, se introdujeron las promesas. Una promesa es un objeto que representa la eventual finalización de una operación asíncrona.

Se crean con new Promise() y se encadenan con then() para manejar el resultado, además del catch() para los errores.

```
const tareaAsincrona = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Tarea completada");
  }, 1000);
});

tareaAsincrona
  .then(resultado => console.log(resultado))
  .catch(error => console.error(error));
```

```
PS C:\Users\Luis Caldera\OneDrive\Escritorio\Seminario Node.js> node promise.js
Tarea completada
```

Callbacks, Promesas y async/await

Lo más usado por tema de legibilidad y control de errores son las funciones async y await. Estas funciones permiten escribir código **asíncrono** que parece **síncrono**.

Una función async puede usar await para esperar a que la promesa se resuelva o se rechace.

asyncAwait.js > obtenerDatos

```
// Definición de la Promesa (que ya hemos visto antes)
const tareaAsincrona = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Datos cargados con éxito.");
    }, 1000);
});

// ¡La función async que usa await!
async function obtenerDatos() {
    console.log("1. Pidiendo datos...");

    try {
        // await pausa la ejecución de esta función hasta que la promesa resuelva
        const resultado = await tareaAsincrona;

        // Esta línea se ejecuta 1 segundo después
        console.log("2. Resultado: " + resultado);

    } catch (error) {
        // try/catch captura el error si usamos reject
        console.error("Error al obtener datos:", error);
    }

    console.log("3. Proceso de obtener datos finalizado.");
}

obtenerDatos();
```

```
PS C:\Users\pabli\Desktop\SeminarioNode> node asyncAwait.js
1. Pidiendo datos...
2. Resultado: Datos cargados con éxito.
3. Proceso de obtener datos finalizado.
```

Introducción al uso módulos

Concepto	Definición
Require()	Es la forma tradicional y dinámica de cargar módulos en Node.js
Import	Es la sintaxis moderna y estática basada en el estándar ES Modules, con ventajas en rendimiento y mantenimiento, pero con ciertas limitaciones.

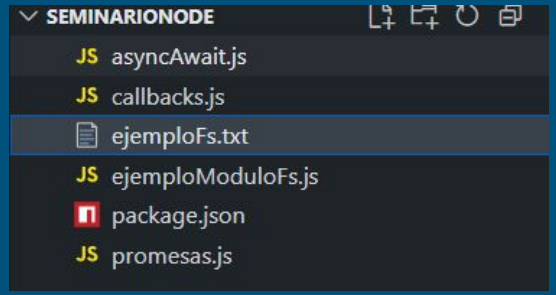
-Módulo fs:

Es un módulo que nos permite interactuar con los archivos de nuestro dispositivo. Al ser un módulo predeterminado de Node.js lo podemos añadir con un require().

Introducción al uso módulos

Ejemplo práctico con el **módulo fs**:

1. Creamos en el workspace un archivo de texto (con el contenido que queramos).



2. Creamos el archivo de node con el siguiente código:

```
JS ejemploModuloFs.js > ...
1   const fs = require('fs');
2   const data = fs.readFileSync('ejemploFs.txt', 'utf8');
3   console.log(data);
```

3. Ejecutamos el archivo desde la terminal.

```
PS C:\Users\pabli\Desktop\SeminarioNode> node ejemploModuloFs.js
Hola usando fs
```

BLOQUE 3: Primer servidor con NodeJS

1- Módulo http.

2- Solicitudes (request) y respuestas (response).

3- Creando nuestro servidor básico.

Módulo http

El módulo http nos permite crear servidores web y manejar solicitudes y respuestas sin necesidad de librerías externas (ya que es un módulo nativo de NodeJS).

Se importa usando el require(‘http’).

```
const http = require('http');
```

Solicitudes (request) y respuestas (response)

-Req: Es un objeto que contiene toda la información que nos envía el cliente (el navegador).

-Res: Es el objeto que usamos para construir la respuesta que le vamos a enviar de vuelta al cliente.

Para construir la respuesta, usamos dos métodos principales de res:

-res.writeHead(código, cabeceras)

-res.end(datos)

Creando nuestro servidor básico

Seminario Node.js > JS crearServidor.js > ...

```
1   const http = require('http');
2
3   const server = http.createServer((req, res) => {
4     console.log("Petición recibida");
5     res.writeHead(200, { 'Content-Type': 'text/plain' });
6     res.end('Primer servidor seminario Node.js');
7   });
8
9   server.listen(3000, () => {
10    console.log('http://localhost:3000');
11  });
```

Con el server.listen le indicamos el puerto

```
PS C:\Users\Luis Caldera\OneDrive\Escritorio\Seminario Node.js> node crearServidor.js
http://localhost:3000
Petición recibida
Petición recibida
Petición recibida
Petición recibida
```

BLOQUE 4: Conexiones con bases de datos SQL

1- Librería SQL.

2- Cómo establecer una conexión.

3- Crear nuestra primera tabla desde Node.

Librería SQL

Abrimos la terminal del proyecto y escribimos el siguiente comando:

```
PS C:\Users\Luis Caldera\OneDrive\Escritorio\Seminario Node.js> npm install mysql

up to date, audited 173 packages in 1s

33 packages are looking for funding
  run `npm fund` for details
```

Cómo establecer una conexión

1- Crear el archivo para establecer la conexión (ej: conexionDB.js)

2- Abrimos XAMPP y creamos una base de datos.

3-

```
const mysql = require('mysql')

const conexion = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password : '',
    database: 'seminario'
})

conexion.connect((error) =>{
    if(error){
        console.error("El error de conexion es: " + error)
        return
    }
    console.log("Conexion con BD con éxito.")
})
```

```
PS C:\Users\pabli\Desktop\SeminarioNode> node conexionDB.js
Conectado con éxito a la base de datos
```

Crear nuestra primera tabla desde Node

Dentro de nuestro archivo de conexión vamos a escribir el siguiente código:

```
const crearTabla = `
    CREATE TABLE IF NOT EXISTS empleados (
        numero INT PRIMARY KEY,
        nombre VARCHAR(50),
        rol VARCHAR(50)
    )
`;

conexion.query(crearTabla, (error, resultado) => {
if (error) {
    console.error("Error al crear la tabla:", error);
    return;
}
console.log("Tabla creada o ya existente.");
});
```

```
PS C:\Users\pabli\Desktop\SeminarioNode> node conexionDB.js
Conectado con éxito a la base de datos
Tabla creada o ya existente
```

BLOQUE 5: Interfaz web con inserción de datos

1- Importar conexión base de datos.

2- Uso de Express y body-parser.

3- Creación del HTML.

4- Qué son req y res.

5- Lógica del formulario.

6- Esperar respuesta del servidor.

Importar conexión base de datos.

Con el archivo de conexión a la base de datos creado en el anterior apartado, haremos un nuevo ejercicio. Para no tener que repetir esa conexión a la base de datos, importamos ese archivo dentro del nuevo que crearemos:

const “nombre variable” = require(‘./ ”nombre archivo conexion” ’);

```
const conexion = require('./conexionBD');
```

Antes de importarlo, necesitaremos exportarlo desde el archivo de conexión para eso haremos en él:

module.exports = “nombre variable con la que se crea la conexión”

```
module.exports = conexion
```

Uso de Express y body-parser

En este nuevo programa haremos uso de 2 módulos más junto a la conexión que ya tenemos. Estos módulos serán **Express** y **body-parser**:

a. Express: crea el servidor web y gestiona las rutas, respuestas y flujos HTTP.
b. body-parser: analiza la información enviada por el cliente en el cuerpo de las peticiones, convirtiéndola en estructura accesible para usar en el backend.

```
const conexion = require('./conexionBD');
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
app.use(bodyParser.urlencoded({ extended: true }));
```

Creación del HTML

¡Comenzamos con la creación del HTML! En el archivo nuevo, copiamos este trozo de código:

```
app.get('/', (req, res) => {
  res.send(`
    <form action="/add" method="POST">
      <input type="text" name="nombre" placeholder="Nombre" required>
      <input type="text" name="rol" placeholder="Rol" required>
      <button type="submit">Añadir</button>
    </form>
  `);
});
```

El método GET se usa para solicitar datos al servidor. Lo que significa que conseguiremos los recursos o páginas que no modifican datos en el servidor.

Qué son req y res

Req y res son los manejadores de rutas de **Express**.

-Req es lo que recibimos del cliente (datos del formulario, URL, cabeceras, etc.).

-Res es lo que enviamos de vuelta al cliente (HTML, JSON, mensajes de éxito, etc.).

Lógica del formulario

Ahora pasamos con la lógica del formulario, donde conseguiremos a través de “req” lo que introducimos por teclado en el formulario, para luego enviar la sentencia SQL a través de un método POST:

Método **POST**: Principal función de enviar datos al servidor, principalmente para crear o actualizar recursos. Se usa en formularios.

```
// Recibir datos del formulario y guardar en MySQL
app.post('/add', (req, res) => {
  const { nombre, rol } = req.body;
  const sql = 'INSERT INTO empleados (nombre, rol) VALUES (?, ?)';

  conexion.query(sql, [nombre, rol], (err, resultado) => {
    if (err) {
      return res.status(500).send('Error al añadir empleado: ' + err.message);
    }
    res.send('Empleado añadido correctamente');
  });
});
```

Esperar respuesta del servidor.

Con el frontend (Formulario HTML) y el backend (Lógica Formulario), está la página terminada. Ahora solo falta enviar una petición al servidor para que ponga en marcha nuestro programa, para ello usaremos:

```
app.listen(3000, () => {
  console.log('Ejecución correcta ahora abre el localhost 3000.');
});
```

El resultado de la página se encontrará aquí: “http://localhost:(Nº puerto)”

BLOQUE 6: Cierre y tarea para casa

1- Ejercicio práctico final (para casa).

Ejercicio práctico final (para casa)

Ejercicio propuesto:

Queremos realizar un HTML donde se pueda agregar datos dentro de la tabla de empleados, como hicimos antes, pero que al añadir un empleado no se quite el formulario y se puedan seguir añadiendo más empleados (lo que sería una web dinámica), el número de empleados deben de asignarse automáticamente, el nombre introducirlo por teclado y el rol asignarlo mediante un desplegable donde contenga estos roles: “User, Admin, Developer” Solo pueden aparecer esos 3 roles y asignarlos a cada usuario.
Aparte de la introducción de datos, tendrás que reflejar justo debajo del formulario una tabla que contendrá los datos de la tabla “empleados”.
Pista: para que a la hora de hacer el programa no recargue la página por cada método POST, GET o DELETE que se hace, investiga sobre las funciones fetch, te ayudarán en este tema. Con esto, conseguirás hacer tu web dinámica.

-Estructura del programa (La estructura no es obligatoria pero podría ser una buena estructura para facilitar el desarrollo y entendimiento del programa.) :
-Server.js: Donde harás las peticiones a la base de datos e insertar datos en la base de datos.
-ConexionBD.js: Aquí tendrá que estar la conexión a la base de datos, con su debido export para posteriormente usarla.
Carpeta FrontEnd:
-index.html: El html deberá contener un formulario y una tabla con sus respectivos campos.
-scripts.js: Es donde se controla que la página sea dinámica, haciendo diversas funciones haciendo uso de fetch().