

Sprint 1 - Proyecto 2 Documentación

Índice

- 1. Introducción a Node.js y sus Módulos Core
 - 2. Módulo 1: `path`
 - 2.1 ¿Por qué es importante el Módulo `path` ?
 - 2.2 Comparativa con otros Modulos
 - 2.3 Funciones Clave del Módulo `path`
 - 3. Módulo 2: `os`
 - 3.1 ¿Por qué es importante el Módulo `os` ?
 - 3.2 Funciones Clave del Módulo `os`
 - 4. Módulo 3: `fs`
 - 4.1 ¿Por qué es importante el Módulo `fs` ?
 - 4.2 Funciones Clave del Módulo `fs`
 - 5. Módulo 4: `events`
 - 5.1 ¿Por qué es importante el Módulo `events` ?
 - 5.2 Uso de la Clase `EventEmitter`
 - 6. Módulo 5: `http`
 - 6.1 Introducción al Módulo `http`
 - 6.2 Creación de un Servidor HTTP Básico
 - 6.3 Manejo de Rutas con Parámetros Dinámicos
 - 6.4 Rutas con Expresiones Regulares
 - 6.5 Manejo de Múltiples Métodos HTTP
 - 6.6 Rutas Anidadas y Manejo de Subrutas con Parámetros
 - 6.7 Manejo de Rutas con Parámetros de Consulta (Query Parameters)
-

1. Introducción a Node.js y sus Módulos Core

¿Qué es Node.js?

Node.js es un entorno de ejecución de JavaScript que permite ejecutar este lenguaje fuera del navegador, principalmente en el lado del servidor. Se construyó sobre el motor **V8** de Google, el cual fue diseñado inicialmente para Google Chrome y es muy eficiente en la ejecución de JavaScript.

Ventajas del Motor V8 y la Naturaleza Asíncrona de Node.js

El motor V8 es rápido, y sumado a la naturaleza no bloqueante y asíncrona de Node.js, hace que sea excelente para manejar múltiples conexiones simultáneas. Node.js sobresale en aplicaciones como **chats en**

tiempo real, APIs REST y juegos multijugador, donde la velocidad y la capacidad de manejar muchas solicitudes concurrentes son críticas.

Arquitectura Asíncrona y No Bloqueante

Node.js utiliza un modelo basado en eventos para la gestión de las operaciones de entrada/salida (I/O). A diferencia de los sistemas tradicionales donde la operación de I/O bloquea la ejecución hasta completarse, en Node.js estas operaciones son no bloqueantes. El flujo de ejecución continúa mientras las operaciones de I/O ocurren en segundo plano.

Diferencia Entre Bloqueante y No Bloqueante

En un entorno tradicional, una operación de lectura de archivo detendría el proceso hasta que el archivo sea completamente leído. Node.js, sin embargo, permite que la ejecución continúe mientras realiza esas operaciones de forma asíncrona, notificando al hilo principal cuando la tarea ha finalizado.

Ejemplo Comparativo:

Código Bloqueante (tradicional):

```
import fs from 'fs';

// Bloquea hasta que la lectura se complete
const data = fs.readFileSync('/path/to/file');
// Se ejecuta después de que la lectura haya terminado
console.log('File read:', data.toString());
```

Código No Bloqueante (Node.js):

```
import fs from 'fs';

fs.readFile('/path/to/file', (err, data) => {
  if (err) throw err;
  // Se ejecuta cuando se completa la lectura
  console.log('File read:', data.toString());
});

console.log('No espera la lectura, sigue ejecutando otras operaciones');
```

En el ejemplo no bloqueante, `fs.readFile()` devuelve el control inmediatamente y continúa ejecutando otras instrucciones mientras la lectura del archivo se realiza en segundo plano.

¿Qué son los Módulos Core en Node.js?

Los módulos Core son paquetes nativos incluidos en Node.js que proporcionan funciones esenciales como la manipulación de archivos, gestión de servidores, interacciones con el sistema operativo, entre otras. No requieren instalación adicional desde NPM, ya que forman parte de la instalación base de Node.js.

Módulos Core vs Módulos de Terceros

- **Módulos Core:**

- Incluidos en Node.js y disponibles en cualquier instalación.
- Ofrecen alto rendimiento, optimización y seguridad.
- Ejemplos: `fs` (sistema de archivos), `http` (servidor HTTP), `path` (manipulación de rutas).

- **Módulos de Terceros:**

- Deben ser instalados mediante NPM.
- Ofrecen funcionalidades adicionales no cubiertas por los módulos Core.
- Ejemplos: **Express** (framework web), **Mongoose** (conexión a bases de datos MongoDB).

Filosofía detrás de los Módulos Core

Node.js sigue una filosofía de mantener un núcleo minimalista. Las funciones avanzadas o específicas se delegan a módulos de terceros para asegurar un entorno flexible y ligero. Esto permite que los desarrolladores elijan e integren solo las herramientas necesarias para su proyecto, manteniendo el entorno optimizado y rápido.

2. Módulo 1: `path`

2.1 ¿Por qué es importante el Módulo `path`?

El módulo `path` permite la manipulación de rutas de archivos y directorios de manera que sea compatible entre diferentes plataformas. Esto es especialmente importante en entornos multiplataforma, como Windows y Linux, donde las convenciones de rutas son diferentes (`\` en Windows vs `/` en Unix).

2.2 Comparativa con Otros Módulos

- `fs`: Se usa para trabajar con el sistema de archivos, pero no está especializado en manipular rutas. Se recomienda usar `path` junto con `fs` para una gestión eficiente de las rutas.
- `url`: El módulo `url` se usa para trabajar con URLs y no es adecuado para la manipulación de rutas del sistema de archivos.

2.3 Funciones Clave del Módulo `path`

`path.dirname()`

Devuelve el directorio de una ruta dada.

```
import path from 'path';

const dir = path.dirname('/usr/local/bin/node');
console.log('Directorio base:', dir); // Salida: '/usr/local/bin'
```

- **Funcionalidad:** Útil cuando se necesita obtener el directorio de un archivo y no la ruta completa.

`path.basename()`

Devuelve el nombre del archivo en una ruta, con o sin su extensión.

```
const base = path.basename('/usr/local/bin/file.txt', '.txt');
console.log('Nombre del archivo:', base); // Salida: 'file'
```

- **Funcionalidad:** Permite extraer el nombre del archivo, excluyendo la extensión si se desea.

`path.extname()`

Devuelve la extensión del archivo.

```
const ext = path.extname('/usr/local/bin/file.txt');
console.log('Extensión del archivo:', ext); // Salida: '.txt'
```

- **Funcionalidad:** Específica para obtener la extensión de un archivo, útil para verificar tipos de archivos.

`path.join()`

Une varios segmentos de ruta en una sola cadena.

```
const fullPath = path.join('/home', 'user', 'docs', 'file.txt');
console.log('Ruta completa:', fullPath); // Salida en Linux: '/home/user/docs/file.txt'
```

- **Funcionalidad:** Garantiza que los separadores de directorio se manejen correctamente según el sistema operativo, evitando errores de concatenación manual.

3. Módulo 2: `os`

3.1 ¿Por qué es importante el Módulo `os`?

El módulo `os` permite interactuar

con el sistema operativo subyacente en el que se ejecuta Node.js, proporcionando información sobre la CPU, memoria, interfaces de red y más. Es esencial para crear aplicaciones que necesiten adaptarse a los recursos del sistema en el que están desplegadas.

3.2 Funciones Clave del Módulo `os`

`os.arch()`

Devuelve la arquitectura de la CPU.

```
import os from 'os';

console.log('Arquitectura de la CPU:', os.arch()); // Salida: 'x64'
```

- **Funcionalidad:** Proporciona la arquitectura del sistema, útil para tomar decisiones basadas en el hardware disponible.

`os.platform()`

Devuelve el nombre del sistema operativo.

```
console.log('Plataforma del sistema:', os.platform()); // Salida: 'linux'
```

- **Funcionalidad:** Permite saber en qué plataforma se está ejecutando el código para adaptarse a diferencias entre sistemas operativos.

`os.totalmem()`

Devuelve la cantidad total de memoria disponible en el sistema.

```
console.log('Memoria total:', os.totalmem(), 'bytes');
```

- **Funcionalidad:** Útil para monitorear los recursos del sistema y optimizar el rendimiento según la memoria disponible.

4. Módulo 3: `fs`

4.1 ¿Por qué es importante el Módulo `fs`?

El módulo `fs` (file system) permite a Node.js interactuar con el sistema de archivos del sistema operativo. Con él, puedes leer, escribir, eliminar y manipular archivos de manera sencilla. Es uno de los módulos más utilizados en aplicaciones que interactúan con datos locales.

4.2 Funciones Clave del Módulo `fs`

`fs.readFile()`

Lee un archivo de manera asíncrona.

```
import fs from 'fs';

fs.readFile('./data/example.txt', 'utf8', (err, data) => {
  if (err) {
    console.error('Error al leer el archivo:', err);
    return;
}
```

```
    console.log('Contenido del archivo:', data);
});
```

- **Funcionalidad:** Permite leer archivos sin bloquear el hilo principal de la aplicación.

fs.writeFile()

Escribe contenido en un archivo de manera asíncrona. Si el archivo no existe, lo crea.

```
fs.writeFile('./data/newfile.txt', 'Este es un nuevo archivo', (err) => {
  if (err) throw err;
  console.log('Archivo escrito exitosamente');
});
```

- **Funcionalidad:** Es útil para guardar datos de manera eficiente sin bloquear la aplicación.

fs.rename()

Renombra un archivo o lo mueve a una nueva ubicación.

```
fs.rename('./data/newfile.txt', './data/renamedfile.txt', (err) => {
  if (err) throw err;
  console.log('Archivo renombrado');
});
```

- **Funcionalidad:** Permite reorganizar archivos en el sistema de manera sencilla.

5. Módulo 4: events

5.1 ¿Por qué es importante el Módulo events?

El módulo `events` en Node.js permite crear y manejar eventos personalizados. La arquitectura basada en eventos de Node.js se utiliza para manejar acciones asíncronas, lo que es fundamental para aplicaciones en tiempo real y altamente concurrentes.

5.2 Uso de la Clase EventEmitter

La clase `EventEmitter` es fundamental para trabajar con eventos en Node.js. Permite emitir eventos y definir funciones que se ejecutarán en respuesta a esos eventos.

EventEmitter.on()

Escucha eventos emitidos.

```
import { EventEmitter } from 'events';
```

```
const emisor = new EventEmitter();

emisor.on('eventoPersonalizado', () => {
  console.log('Evento personalizado recibido');
});

emisor.emit('eventoPersonalizado'); // Emitir el evento
```

- **Funcionalidad:** `on()` permite definir un manejador que se ejecutará cada vez que el evento especificado sea emitido.
-

6. Módulo 5: http

6.1 Introducción al módulo http (Servidor HTTP y Manejo de Rutas)

El módulo `http` de Node.js permite la creación de servidores HTTP básicos y la gestión de solicitudes (`requests`) y respuestas (`responses`) web. Es especialmente útil para aprender los fundamentos de HTTP antes de trabajar con frameworks más complejos como Express.js.

6.2 Creación de un Servidor HTTP Básico

1. Importar el módulo http

```
import http from 'http'; // Importamos el módulo HTTP para crear el servidor.
```

2. Creación de un Servidor HTTP Básico

Un servidor HTTP en Node.js escucha solicitudes entrantes, las procesa y devuelve una respuesta. En este caso, el servidor responde con un mensaje en texto plano.

```
// Creamos un servidor HTTP
const servidor = http.createServer((req, res) => {
  res.statusCode = 200; // Establecemos el código de estado HTTP a 200 (OK)
  // Indicamos que el contenido es texto plano
  res.setHeader('Content-Type', 'text/plain');
  res.end('¡Hola, Mundo!'); // Enviamos una respuesta al cliente
});

// El servidor escucha en el puerto 3000
servidor.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

Explicación:

- `createServer`: Crea el servidor.

- `res.statusCode`: Define el código de estado HTTP de la respuesta (200 = OK).
 - `res.setHeader`: Define las cabeceras HTTP, en este caso, para indicar que se enviará texto plano.
 - `res.end`: Finaliza la respuesta enviando el cuerpo “¡Hola, Mundo!”.
-

6.3 Manejo de Rutas con Parámetros Dinámicos

El manejo de rutas dinámicas es crucial para aplicaciones web que gestionan información en base a identificadores, como usuarios o productos.

```
const servidor = http.createServer((req, res) => {
  const partesUrl = req.url.split('/');
  // Dividimos la URL en partes

  if (partesUrl[1] === 'usuario' && partesUrl[2]) {
    const usuarioId = partesUrl[2];
    // Obtenemos el ID del usuario desde la URL
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    // Enviamos la respuesta con el ID del usuario
    res.end(`Información del usuario con ID: ${usuarioId}`);
  } else {
    res.statusCode = 404;
    res.setHeader('Content-Type', 'text/plain');
    res.end('Ruta no encontrada');
  }
});

servidor.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

Explicación:

- `req.url.split(' / ')`: Divide la URL en partes.
 - Verificamos si la primera parte de la URL es “usuario” y si la segunda parte contiene un ID.
 - Si la ruta es válida, respondemos con el ID del usuario. Si no, enviamos un mensaje de error 404.
-

6.4 Rutas con Expresiones Regulares

Las expresiones regulares permiten manejar rutas de manera flexible y validar parámetros de la URL.

```
const servidor = http.createServer((req, res) => {
  // Definimos una expresión regular para capturar el ID del producto
  const rutaRegex = /^\/producto\/\/(\d+)$/;
  // Intentamos hacer coincidir la URL con la expresión regular
  const match = req.url.match(rutaRegex);

  if (match) {
    const productoId = match[1];
    // Extraemos el ID del producto desde la URL
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
  }
});
```

```

        // Enviamos la respuesta con el ID del producto
        res.end(`Detalles del producto con ID: ${productId}`);
    } else {
        res.statusCode = 404;
        res.end('Ruta no encontrada'); // Enviamos un 404 si la ruta no coincide
    }
});

servidor.listen(3000, () => {
    console.log('Servidor escuchando en el puerto 3000');
});

```

Explicación:

- `rutaRegex`: Captura cualquier URL que comience con `/producto/` seguido de un número (`\d+`).
 - `req.url.match`: Intenta hacer coincidir la URL con la expresión regular. Si coincide, obtenemos el ID del producto.
-

6.5 Manejo de Múltiples Métodos HTTP (GET y POST)

Un servidor HTTP puede manejar varios tipos de solicitudes (métodos HTTP), como `GET` para obtener datos y `POST` para enviar datos.

```

const servidor = http.createServer((req, res) => {
    if (req.method === 'GET' && req.url === '/productos') {
        res.statusCode = 200;
        res.setHeader('Content-Type', 'application/json');
        // Enviamos una lista de productos en formato JSON
        res.end(JSON.stringify([{ id: 1, nombre: 'Producto A' },
            { id: 2, nombre: 'Producto B' }]));

    } else if (req.method === 'POST' && req.url === '/productos') {
        let cuerpo = ''; // Inicializamos una variable para capturar los datos
        // Capturamos los datos recibidos en el cuerpo de la solicitud
        req.on('data', (chunk) => { cuerpo += chunk.toString(); });
        req.on('end', () => {
            // Convertimos el cuerpo a objeto JSON
            const nuevoProducto = JSON.parse(cuerpo);
            res.statusCode = 201;
            res.setHeader('Content-Type', 'application/json');
            res.end(JSON.stringify({ mensaje: 'Producto creado', producto: nuevoProducto }));
        });
    } else {
        res.statusCode = 404;
        res.end('Ruta o método no encontrado');
    }
});

servidor.listen(3000, () => {
    console.log('Servidor escuchando en el puerto 3000');
});

```

Explicación:

- **GET:** Devuelve una lista de productos en formato JSON.
 - **POST:** Recibe un producto en el cuerpo de la solicitud y lo devuelve como respuesta junto con un mensaje de confirmación.
-

6.6 Rutas Anidadas y Manejo de Subrutas con Parámetros

Para aplicaciones más complejas, como aquellas que manejan relaciones de datos (usuarios con pedidos), es útil manejar rutas anidadas.

```
const servidor = http.createServer((req, res) => {
  // Capturamos el ID del usuario y del pedido en la URL
  const rutaRegex = `/^\/usuarios\/(\d+)\/pedidos\/(\d+)/`;
  const match = req.url.match(rutaRegex);

  if (match) {
    const usuarioId = match[1]; // Capturamos el ID del usuario
    const pedidoId = match[2]; // Capturamos el ID del pedido
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/plain');
    res.end(`Detalles del pedido ${pedidoId} para el usuario ${usuarioId}`);
  } else {
    res.statusCode = 404;
    res.end('Ruta no encontrada');
  }
});

servidor.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

Explicación:

- El servidor captura las rutas anidadas como `/usuarios/:usuarioId/pedidos/:pedidoId`, extrayendo ambos parámetros y devolviendo una respuesta adecuada.
-

6.7 Manejo de Rutas con Parámetros de Consulta (Query Parameters)

Los parámetros de consulta (`?clave=valor`) son útiles para pasar datos adicionales en las solicitudes HTTP, por ejemplo, en búsquedas o filtros.

```
import url from 'url';

const servidor = http.createServer((req, res) => {
  // Parseamos la URL para obtener el pathname y los parámetros de consulta
  const parsedUrl = url.parse(req.url, true);
  const pathname = parsedUrl.pathname;
  const query = parsedUrl.query; // Obtenemos los parámetros de consulta

  if (pathname === '/buscar' && req.method === 'GET') {
```

```
const termino = query.q; // Obtenemos el término de búsqueda
res.statusCode = 200;
res.setHeader('Content-Type', 'text/plain');
res.end(`Resultados de búsqueda para: ${termino}`);
} else {
  res.statusCode = 404;
  res.end('Ruta no encontrada');
}
});

servidor.listen(3000, () => {
  console.log('Servidor escuchando en el puerto 3000');
});
```

Explicación:

- `url.parse(req.url, true)`:

Analiza la URL y extrae los parámetros de consulta en formato clave-valor.

- En este caso, buscamos la clave `q` en la URL y la usamos para devolver los resultados de búsqueda.
-