<codoa codo/>

FULL STACK FRONTEND
Clase 26

Node 2

BA

Les damos la bienvenida

Vamos a comenzar a grabar la clase

Clase 25

Node – Introducción

- Arquitectura
- Manipulación de la respuesta recibida

Clase 26

Node – Gestores de paquetes

- Módulos
- Gestores de Paquetes
- Creación de un servidor
- Implementaciones de Node

Clase 27

Node - Express

- Express
- Servidor estático
- Nodemon
- Rutas

MÓDULOS

Uno de los problemas de Javascript desde sus inicios es organizar de una forma adecuada una aplicación grande, con muchas líneas de código.

Tener todo el código en un sólo archivo Javascript es confuso y complejo.

La solución a ese problema fueron los módulos, que permiten separar nuestro código sin tener que vincular una etiqueta script por cada archivo de Javascript en nuestro proyecto.

En Node.js un módulo es simplemente un archivo .js que puede importar o exportar funcionalidades para ser utilizadas en otros archivos. Este archivo puede contener variables, funciones, clases u objetos que pueden ser exportados e importados.

Historia de los módulos

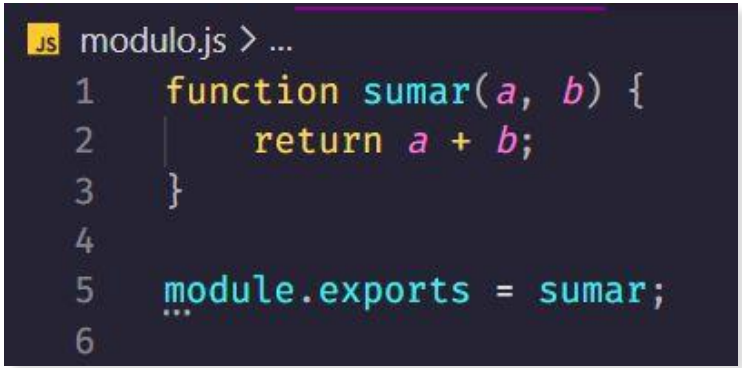
El precursor para esta solución fue **NodeJS** quién creó el sistema de módulos conocido como ***CommonJS***.

Sin embargo, a partir de la especificación EcmaScript 2015 se introduce al lenguaje una alternativa nativa conocida como ***ES Modules***.

Ambos tienen su propia sintaxis y si bien no es común verlos en proyectos con Javascript puro, son muy usados en el Frontend en Frameworks como React o en desarrollos backend realizados con NodeJS.

Veamos un ejemplo de CommonJS

Tenemos un **módulo** con una función que permite sumar 2 números a la cual exportamos con module.exports



Veamos un ejemplo de CommonJS

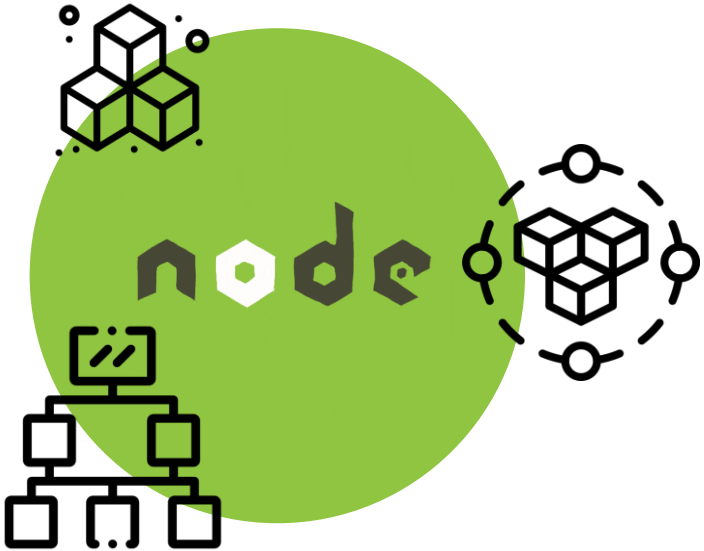
Ahora llamamos a la función sumar del archivo modulo.js y lo utilizamos en nuestro archivo index.js

Tipos de Módulos

Nativos: Son módulos que vienen incluidos dentro del código fuente de NODE.

Internos: Todos los módulos creados durante el desarrollo del proyecto y que hacen a nuestra aplicación.

Externos o de terceros: librerías creadas por terceros y puestas a disposición a través de gestores de paquetes de NODE como **NPM** o **YARN**.



Gestores de Paquetes

Los pakage manager más conocidos para **NODE** son **NPM** y **YARN**.

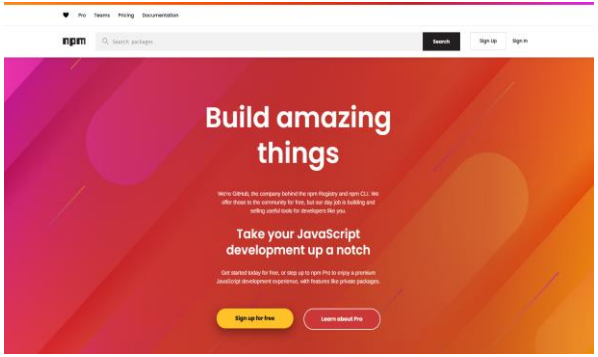
Los gestores de paquetes en Node.js son herramientas que facilitan la instalación, actualización, gestión y eliminación de paquetes (bibliotecas, módulos, dependencias) en un proyecto de Node.js.

Con ellos podemos instalar librerías de código que nos ayuden con tareas simples como animaciones, alertas o trabajo con fechas, sin embargo también podemos usarlo para descargar frameworks como React o Angular.

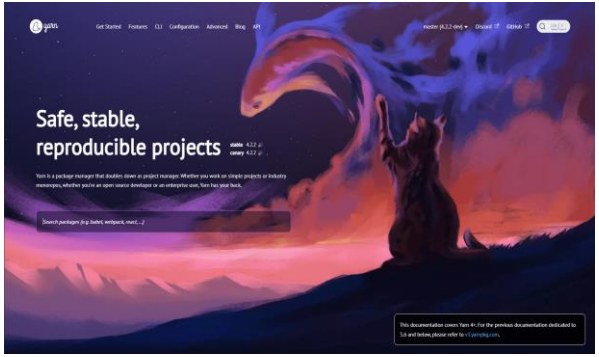
Gestores de paquetes

Cada uno de estos administradores de paquetes tienen sus propias ventajas y desventajas. La elección del administrador de paquetes dependerá de las necesidades específicas del proyecto y las preferencias del equipo de desarrollo.

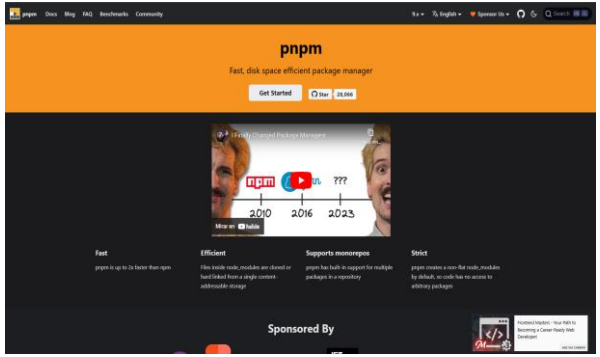
https://www.npmjs.com/



https://yarnpkg.com/



https://pnpm.io/



Agencia de Aprendizaje a lo largo de la vida

NPM

Node Package Manager es el gestor de paquetes más conocido y utilizado.

Se instala automáticamente cuando instalamos **NODE** por lo que **no** debemos instalar nada adicional.

Para poder instalar dependencias o librerías en nuestros proyectos, primero hay que utilizar el comando **NPM init** o **NPM init -y** en la terminal para dar inicio a un proyecto de **NODE** gestionado por **NPM**.

```
>> node  03:06 npm init -y
Wrote to C:\Users\pol_m\Desktop\node\package.json:

{
  "name": "node",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Características Principales de NPM

1.Gestión de Paquetes:

- **Instalación**: Permite instalar paquetes y módulos que otros desarrolladores han creado y compartido.
- **Actualización**: Facilita la actualización de paquetes a sus versiones más recientes.
- **Desinstalación**: Permite eliminar paquetes que ya no son necesarios.

2.Gestión de Dependencias:

- NPM se encarga de instalar todas las dependencias necesarias para que un proyecto funcione correctamente, según lo especificado en el archivo **package.json**.

3.Repositorio Centralizado:

- NPM tiene un repositorio en línea (https://www.npmjs.com/) donde se pueden encontrar miles de paquetes gratuitos y de código abierto para diversas funcionalidades.

4.Scripts de NPM:

- Permite definir scripts personalizados en el archivo **package.json** para automatizar tareas comunes, como iniciar el servidor, ejecutar pruebas, y más.

El archivo package.json

Es un archivo de configuración crucial en cualquier proyecto Node.js. Sirve como la raíz del proyecto. Es un archivo en formato JSON que proporciona información necesaria tanto para el administrador de paquetes NPM como para los desarrolladores que trabajan en el proyecto. Proporciona una forma estructurada de definir y gestionar las dependencias, scripts, metadatos y otras configuraciones importantes del proyecto. Su correcta configuración y uso facilitan el desarrollo, mantenimiento y colaboración en proyectos Node.js.

```json
{
  "name": "mi-proyecto",
  "version": "1.0.0",
  "description": "Este es un proyecto de ejemplo",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

El archivo package.json

name: Es el nombre del proyecto. Este nombre debe ser único si deseas publicar el paquete en el registro de npm.

version: Define la versión actual del proyecto siguiendo el esquema de versionado semántico (semver). La versión se compone de tres partes: mayor, menor y parche (major.minor.patch).

description: Es una breve descripción del proyecto. Es útil proporcionar información sobre el propósito o funcionalidad del proyecto.

main: Especifica el punto de entrada principal de la aplicación. Cuando alguien requiere el módulo, este archivo será el primero en ser cargado.

scripts: Es un objeto que contiene comandos de script que pueden ser ejecutados usando npm run “script-name”.

El archivo package.json

test: Define un comando para ejecutar pruebas. En el ejemplo imprime un mensaje de error y devuelve un código de salida 1, indicando que no hay pruebas especificadas. Este es un valor por defecto que suele reemplazarse con un comando de prueba real, como puede ser con Mocha o Jest.

keywords: Es una lista de palabras clave que ayudan a identificar el proyecto en el registro de npm. En este ejemplo, está vacío, pero normalmente incluirías términos relacionados con el proyecto para mejorar la búsqueda.

author: Especifica el autor del proyecto.

license: Define la licencia bajo la cual se distribuye el proyecto. "ISC" es una licencia de software libre y permisiva. Es importante especificar una licencia para que otros sepan cómo pueden utilizar el código creado.

¡Ahora sí!

A crear un servidor web

¿Qué es un Servidor Web?

Un servidor de **software** en términos generales, se refiere a cualquier programa o aplicación que proporciona servicios o recursos a otros dispositivos o programas en una red (ftp, mysql, tomcat).

Específicamente un **Servidor Web HTTP,** es un servidor de software controla cómo los usuarios de la web obtienen acceso a los archivos alojados en un servidor de hardware (una pc).

Son **capaces de comprender urls** o solicitudes a través de ellas y dar una respuesta atendiendo dicha solicitud.

Existen servidores estáticos y dinámicos.

Tipos de servidores WEB

Estático

Consiste en una computadora (hardware) con un servidor HTTP (software).

Se le dice “estático” porque envía los archivos que aloja “tal como se encuentran” (sin modificarlos) a tu navegador.

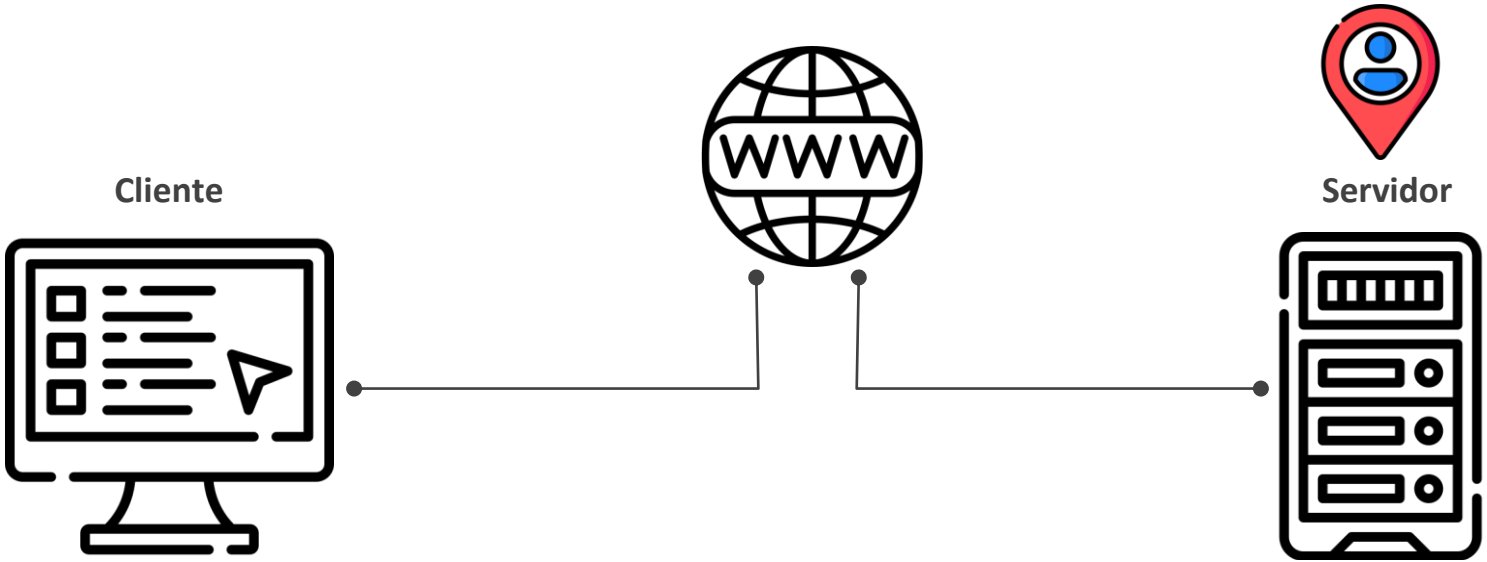
Dinámico

Consiste en un servidor web estático con software adicional, como una aplicación de servidor y una base de datos.

Se le dice “dinámico” a este servidor porque la aplicación actualiza los archivos alojados, antes de enviar el contenido a tu navegador mediante el servidor HTTP.

Para entender un poco más, creemos nuestro primer servidor estático con 

Premisa fundamental



Servidor Estático Nativo

- Para poder montar nuestro servidor, debemos tener un archivo de entrada o “entry point” donde realicemos la configuración inicial de nuestro Server.
- Creamos nuestro entry point llamado: **app.js**
- Luego importamos el módulo **http** nativo y creamos un servidor con el método: **.createServer();**


```
//Importación del modulo http
const http = require('http');

//Creación del servidor estático
const server = http.createServer();
```

Servidor Estático Nativo

- Cada llamada a nuestro server recibe 2 parámetros importantes: **req** (require) y **res** (response), que contienen **toda la información** tanto de la solicitud como de la respuesta en ese orden.
- Finalmente, escribimos una cabecera mediante **.writeHead()**; indicando el **tipo de contenido** que vamos a devolver y lo enviamos.

```
//Importación del modulo http
const http = require('http');

//Creación del servidor estático
// Función de devolución de llamada que se
ejecuta cuando se recibe una solicitud
const server = http.createServer((req, res)=>{
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('Hola mundo');
});
```

Servidor Estático Nativo

- Ya casi lo tenemos, ahora solo nos queda **escuchar a un puerto** para poder realizar llamadas a nuestro servidor.
- Para eso usamos el método **.listen();** el cual trabaja sobre el objeto **server** y recibe el puerto como primer parámetro y **una callback** como segundo.

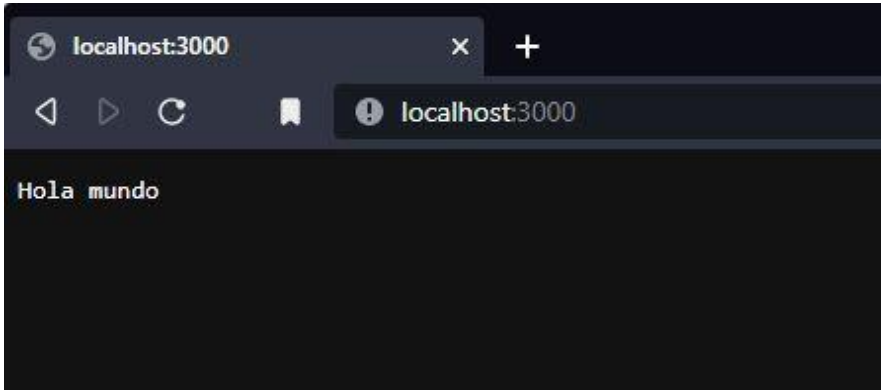
```
const http = require('http');


const server = http.createServer((req, res) => {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    res.end('Hola mundo');
});

//llamando a la func server
server.listen(3000, () => console.log('Servidor
corriendo en puerto http://localhost:3000'));
```

Servidor Estático Nativo

```
>> noder_server  02:02 node app.js
Servidor corriendo en puerto http://localhost:3000
```

¡Listo!

Ahora pongamos a correr nuestro servidor desde la terminal mediante el comando node app.js y accedemos a él desde el navegador.



En este caso devolvemos texto plano, pero intentemos con HTML.

Enviando HTML

Modificamos ligeramente la cabecera Content-Type.

Especificamos charset = ‘UTF-8’, se trata del seteo que reconoce caracteres especiales del dialecto latino.

Además en lugar de texto, enviamos código HTML válido.

```
res.writeHead(200, {
    'Content-Type': 'text/html; charset=UTF-8'
});

res.end('<h1>Hola mundo!</h1><p>Párrafo</p>');
```

Enviando HTML

Reiniciamos el server y volvemos al navegador:

Intentemos con un archivo .html

Leyendo archivos con FileSystem

FileSystem es un módulo nativo de node que **nos permite trabajar con archivos** que existan en la PC o servidor.

Veamos cómo utilizarlo para devolver un archivo HTML como respuesta.

.readFileSync(); lee un archivo de forma síncrona (bloqueante) y luego lo devolvemos como respuesta a la petición.

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

const fs = require('fs');

const server = http.createServer((req, res) => {

    const file = fs.readFileSync(___dirname + '/index.html');

    res.writeHead(200, {
        'Content-Type': 'text/html; charset=UTF-8'
    });

    res.end(file);
});

//Lamamos a la función
server.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

Algunos ejemplos de Implementaciones de Node.js

Cómo ya vimos, Node.js es una plataforma popular para construir aplicaciones de servidor debido a su arquitectura de entrada/salida no bloqueante y su enfoque en el uso de JavaScript en el lado del servidor.

Los ejemplos a continuación muestran cómo Node.js puede ser utilizado para implementar diversas aplicaciones de servidor, desde un simple servidor HTTP hasta una API RESTful y un servidor WebSocket. Cada implementación resalta la flexibilidad y el poder de Node.js para manejar diferentes tipos de cargas de trabajo en aplicaciones modernas.

1. Servidor HTTP Básico

Un servidor HTTP básico es un programa que maneja solicitudes HTTP (Hypertext Transfer Protocol) y envía respuestas HTTP a los clientes, generalmente navegadores web. Este tipo de servidor escucha en un puerto específico, procesa las solicitudes entrantes y devuelve los recursos solicitados o una respuesta adecuada.

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

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

const port = 3000;
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

2. Servidor con Express

Express es un marco de aplicación web para Node.js, diseñado para facilitar la creación de aplicaciones web y API robustas y eficientes. Es minimalista, flexible y está repleto de características que permiten a los desarrolladores crear aplicaciones web y servicios con facilidad.

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

app.get('/', (req, res) => {
  res.send('Hello World');
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

3. API RESTful con Express

Un ejemplo de una API RESTful básica que maneja operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para una colección de "usuarios".

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

let users = [
  { id: 1, name: 'John Doe' },
  { id: 2, name: 'Jane Smith' }
];

app.use(express.json());

// Obtener todos los usuarios
app.get('/users', (req, res) => {
  res.json(users);
});

// Obtener un usuario por ID
app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).send('User not found');
  res.json(user);
});
```

```
// Crear un nuevo usuario
app.post('/users', (req, res) => {
  const user = {
    id: users.length + 1,
    name: req.body.name
  };
  users.push(user);
  res.status(201).json(user);
});

// Actualizar un usuario existente
app.put('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  if (!user) return res.status(404).send('User not found');

  user.name = req.body.name;
  res.json(user);
});

// Eliminar un usuario
app.delete('/users/:id', (req, res) => {
  const userIndex = users.findIndex(u => u.id === parseInt(req.params.id));
  if (userIndex === -1) return res.status(404).send('User not found');

  users.splice(userIndex, 1);
  res.status(204).send();
});

app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

4. Servidor WebSocket

Es un programa que maneja conexiones WebSocket, permitiendo la comunicación bidireccional en tiempo real entre un cliente y un servidor. A diferencia de HTTP, que es un protocolo sin estado y de solicitud-respuesta, WebSocket mantiene una conexión abierta persistente que permite la transferencia de datos en ambas direcciones sin la sobrecarga de las solicitudes HTTP repetidas.

```
const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 8080 });

server.on('connection', socket => {
  console.log('Client connected');
  socket.on('message', message => {
    console.log(`Received: ${message}`);
    socket.send(`Echo: ${message}`);
  });

  socket.on('close', () => {
    console.log('Client disconnected');
  });
});

console.log('WebSocket server running at ws://localhost:8080/');
```

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto