

Agencia de  
Aprendizaje  
a lo largo  
de la vida

# Desarrollo Fullstack



# Les damos la bienvenida

Vamos a comenzar a grabar la clase

## Clase 24

### Node JS

- ▶ ¿Qué es?
- ▶ ¿Cómo funciona?
- ▶ Single Thread vs Multi Thread
- ▶ Instalación

## Clase 25

### Node JS

- ▶ Módulos
- ▶ Node Package Manager
- ▶ Servidor Web Node Nativo
- ▶ Enviar Texto
- ▶ Enviar Archivos

## Clase 26

### Express JS

- ▶ Express JS
- ▶ Express Generator
- ▶ Rutas

# Sigamos con Node!



# NODE JS

Web Server



# En esta clase vamos a aprender a **crear un** **servidor web** con **NODE**

**Pero antes ...**

**Algunos conceptos nuevos.**

# 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.



# 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`

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

```
JS modulo.js > ...  
1  function sumar(a, b) {  
2      return a + b;  
3  }  
4  
5  module.exports = sumar;  
6
```

\*más adelante veremos como es la sintaxis para manejar módulos con ES Modules. Vamos despacio 😊

```
JS index.js > ...  
1  const sumar = require('./modulo');  
2  
3  const resultado = sumar(10, 17);  
4  
5  console.log(resultado);
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
● >> node 02:46 node index.js  
27  
○ >> node 02:46
```

# 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 **package manager** más conocidos para **NODE** son **NPM** y **YARN**.

Estos sirven como bibliotecas que contienen módulos de terceros con soluciones sencillas a problemas comunes y a veces no tan comunes.

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.

# 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"
}
```

# package.json

Al iniciar **NPM** en nuestro proyecto se crea un archivo llamado **package.json**.

Este archivo será utilizado por el gestor de paquetes para listar nuestras dependencias a medida que las vayamos instalando y para guardar información general del proyecto como:

- **Nombre**
- **Versión**
- **Descripción**
- **Autor**
- **Licencia**

```
package.json > ...
1  {
2    "name": "node",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC"
12 }
13
```

La propiedad **main** indica el **archivo de “entrada”** de nuestro proyecto mientras que la propiedad **scripts** nos permite **crear comandos para ejecutar distintas acciones** de nuestro código.

# ¡Ahora sí!

## A crear un servidor web

# ¿Qué es un Servidor Web?

Un servidor de **software** o **Servidor Web HTTP** 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 

# 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();
```



```
const http = require('http');  
  
const server = http.createServer();
```

# Servidor Estático Nativo

Cada llamada a nuestro server recibe 2 parámetros super importantes: `require` y `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.

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

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 nuestra constante `server` y **recibe el puerto** como primer parámetro y **un callback** en segundo.

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

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

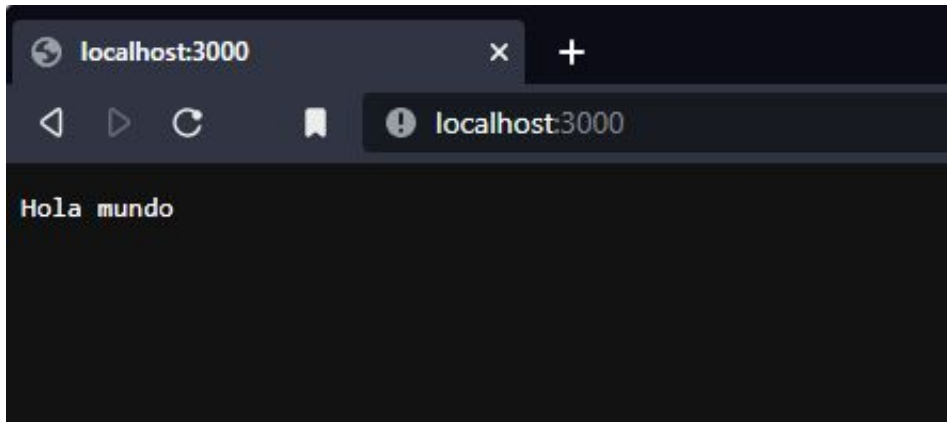
server.listen(3000, () => console.log('Servidor
corriendo en puerto http://localhost:3000'));
```

# Servidor Estático Nativo

## ¡Listo!

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

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



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

# Enviando HTML

Modificamos ligeramente la cabecera **Content-Type**.

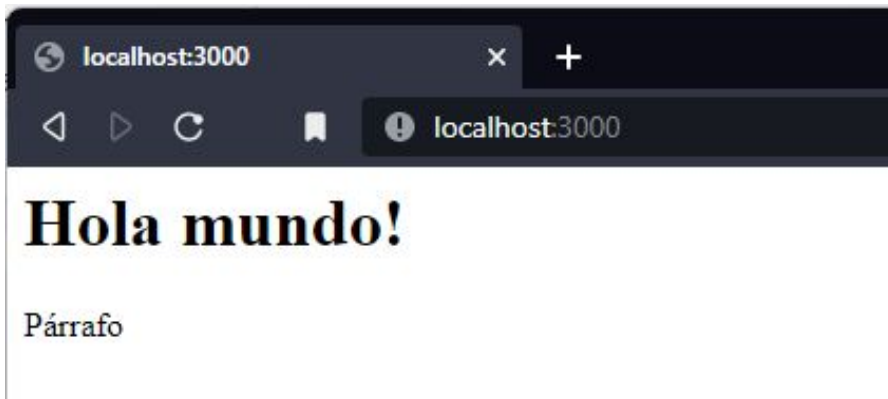
*Si no indicamos el charset, el tilde de "párrafo" no se verá correctamente.*

```
res.writeHead(200, {  
  'Content-Type': 'text/html; charset=UTF-8'  
});  
  
res.end('<h1>Hola mundo!</h1><p>Párrafo</p>');
```

Además en lugar de texto, enviamos **código HTML** válido.

Reiniciamos el server y volvemos al navegador:

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





# 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.

```
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);
});
```

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

# No te olvides de dar el presente

## **Recordá:**

- **Revisar la Cartelera de Novedades.**
- **Hacer tus consultas en el Foro.**

**Todo en el Aula Virtual.**

# Gracias