

«Talento Tech»

Back-End

Node JS

Clase 05



Clase N° 5 - Módulos y gestores de paquetes.

Temario:

1. **Gestores de paquetes:**
 - ¿Qué son? ¿Para qué sirven?
2. **NPM - Node Package Manager:**
 - Iniciar proyecto con Node Js y NPM
 - Instalación de paquetes y gestión de dependencias
 - Creación de scripts
3. **Módulos**
 - ¿Qué son los módulos?
 - Importando, exportando y creando módulos
4. **Dirname**
 - Gestión de rutas absolutas y acceso a archivos del servidor

Objetivos de la Clase.

En esta clase, los estudiantes comprenderán qué son los gestores de paquetes y cómo utilizarlos en proyectos con Node.js. Aprenderán a iniciar y configurar un proyecto utilizando Node.js y NPM, abarcando la instalación de dependencias y la creación de scripts. Además, explorarán los módulos nativos como **path**, **fs**, y **process** para manejar rutas, archivos y procesos. También implementarán módulos de terceros en un proyecto mediante NPM y aprenderán a manejar rutas absolutas y relativas utilizando **__dirname** para acceder a archivos del servidor de manera eficiente.

Gestores de paquetes.

Definición.

Imaginá una biblioteca, donde se guardan infinita cantidad de libros de distintos autores que contienen información determinada. De pronto necesitás saber sobre un tema en especial y vas a la biblioteca en búsqueda de un libro que aporte esa información.

Un gestor de paquetes funciona como una biblioteca de módulos o paquetes de código de programación donde a su vez eliminan la necesidad de descargar e integrar manualmente estas bibliotecas, lo que simplifica el desarrollo y asegura que las dependencias estén organizadas y actualizadas.

Estas herramientas que facilitan la instalación, actualización y gestión de bibliotecas o dependencias en un proyecto. Estas dependencias son piezas de código que otros desarrolladores han creado para resolver problemas comunes, como conectarse a una base de datos, realizar peticiones HTTP o manejar archivos.

¿Para qué sirven los gestores de paquetes?

- Instalar dependencias de terceros.
- Gestionar versiones de esas dependencias.
- Actualizar y eliminar dependencias de forma sencilla.
- Resolver la integración e interacción entre las distintas dependencias y las versiones de las subdependencias que tienen instaladas.

El concepto de dependencias, paquetes o módulos es propio de la programación en general, incluso existen gestores de paquetes para instalar programas en los distintos sistemas operativos. Conozcamos algunos de ellos:

NPM

Node Package Manager o gestor de paquetes de Node JS, podría considerarse el más conocido y utilizado, sin embargo existen alternativas como **yarn** o **pnpm**. Cuanto más alcance tiene un gestor de paquetes más probable es que la dependencia o librería que estés necesitando haya sido subida ahí por el desarrollador.



Homebrew

Permite manejar programas del sistema operativo Mac OS presente en las computadoras marca Apple.



Composer

Usado en proyectos del lenguaje de programación **PHP**.



Chocolatey

El gestor de paquetes del sistema operativo Microsoft nos permite instalar infinidad de programas desde la terminal sin tener que descargar los ejecutables.



Pip

Gestor de paquetes para el lenguaje **Python**.

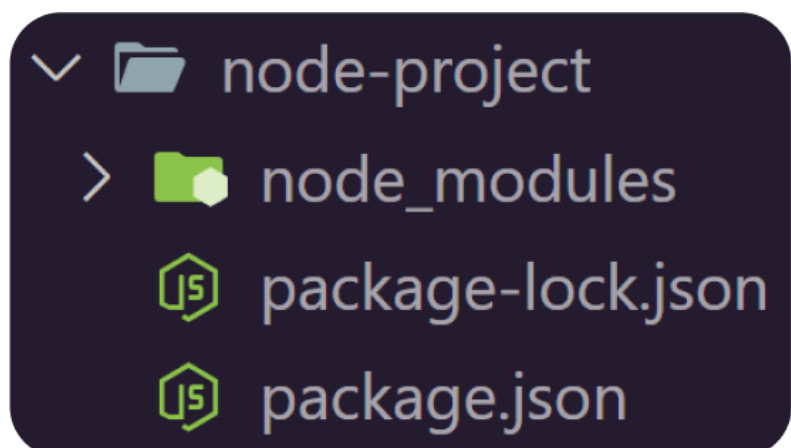
Con los gestores de paquetes, podemos obtener enormes ventajas desde el acceso a miles de librerías o módulos de terceros hasta un entorno estandarizado de configuración. 🚀

NPM - Node Package Manager:

Anteriormente te adelantamos que Node Package Manager nos permite gestionar librerías y código de terceros en nuestros proyectos, sin embargo, tiene muchos más poderes de los que podemos imaginar. Además de gestionar paquetes, **NPM** se encarga de listar y establecer el orden en el que se instala cada dependencia y a su vez cuando estas dependencias utilizan subdependencias, es decir, que el código de origen también utilizó **NPM** para instalar paquetes, el gestor se encarga de decidir qué versión de cada una de ellas debe utilizar para no entrar en conflictos.

Como si esto fuera poco **NPM** crea de forma automática un archivo llamado **package-lock.json** con el detalle minucioso de esta organización de paquetes y también otro archivo llamado

package.json donde tenemos un detalle directo de las dependencias que nosotros instalamos en el proyecto pero a su vez permite declarar: nombre del proyecto, versión, archivo de inicio, autor, licencia y scripts de ejecución del proyecto, entre otras cosas.



Iniciar proyecto con Node Js y NPM

Comenzar un nuevo proyecto en Node JS es relativamente sencillo, basta con asegurarnos de ejecutar los siguientes pasos:

1. Verifica que tengas instalado **Node JS** y **NPM** usando los siguientes comandos:

```

>> ~ node -v
v20.16.0
>> ~ npm -v
10.8.1
  
```

2. Crea un nuevo directorio o carpeta para tu proyecto.
3. Ingresa a la ubicación del proyecto desde la terminal
4. Ejecuta el comando **npm init** donde la terminal comenzará a pedirnos que completemos determinada información o **npm init -y** para realizar un inicio rápido que nos permita configurar nuestro entorno de trabajo luego.

```

>> node-project npm init

package name: (project) my-node-project
version: (1.0.0)
description: Este es una descripción de ejemplo
entry point: (index.js)
test command:
git repository:
keywords: Node, Javascript
author: Pepe
license: (ISC)
  
```

```

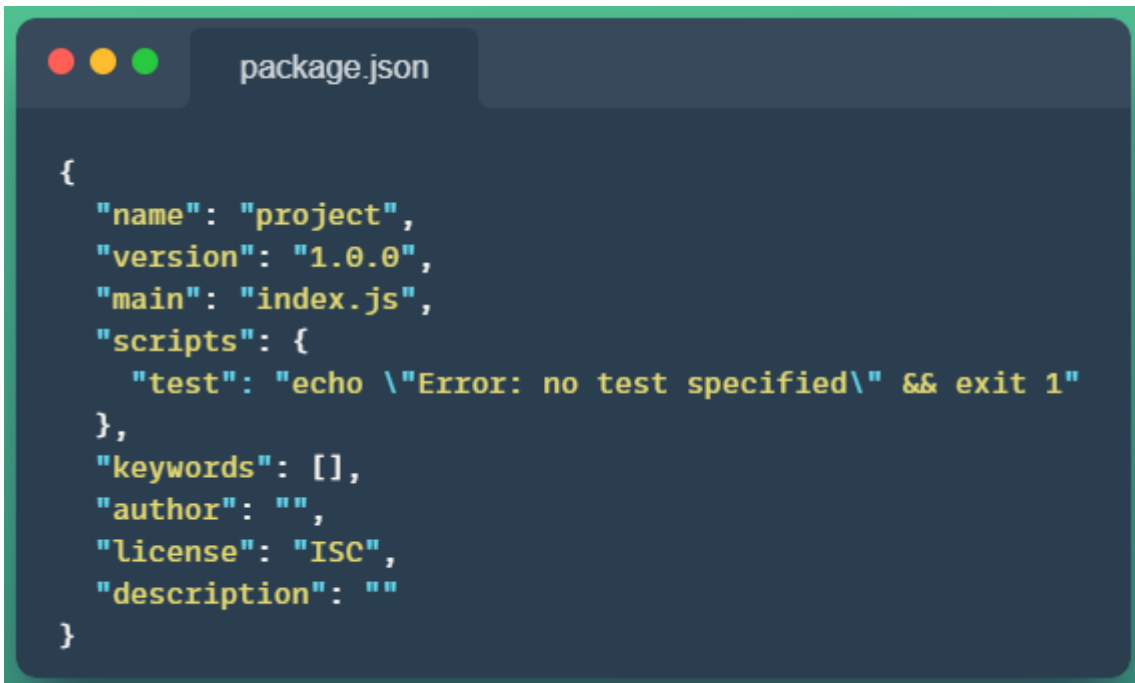
About to write to
C:\your_pc_location\node-project\package.json:

{
  "name": "my-node-project",
  "version": "1.0.0",
  "description": "Este es una descripción de ejemplo",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [
    "Node",
    "Javascript"
  ],
  "author": "Pepe",
  "license": "ISC"
}

Is this OK? (yes)
  
```

En caso de utilizar el flag **-y** salteamos todas las preguntas y podremos cambiar los valores por defecto de cada una de las propiedades en el archivo **package.json** que fue creado como resultado de ejecutar este comando.

Este archivo llamado **package.json** se ve de la siguiente manera:



```

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

- **name:** define el nombre del proyecto. No puede contener guiones medios (-) ni mayúsculas o caracteres especiales.
- **version:** establece el número de versión según la convención **X.Y.Z** donde **X** representa las versiones “*major*” donde existen cambios importantes de nuestro proyecto, la **Y** las versiones “*minor*” con cambios menores y la **Z** las versiones “*patch*” con parches de seguridad o arreglos de errores.
- **main:** indica el “*entry point*” o archivo de entrada de nuestra aplicación.
- **scripts:** propiedad de suma importancia donde podremos crear diferentes scripts para ejecutar nuestro código.
- **keywords:** array de strings donde se colocan palabras clave que definen al proyecto.



- **author:** responsable/s del proyecto.
- **license:** tipo de licencia de uso y explotación del código fuente y/o del proyecto.
- **description:** cadena de texto que describe el proyecto.

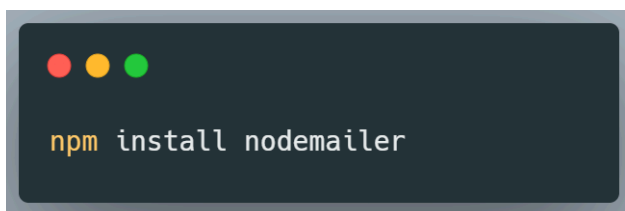
Instalación de paquetes.

Imaginá que estás trabajando en un nuevo proyecto en Node.js y surge la necesidad de enviar correos electrónicos. Al investigar, descubrís que Node.js no incluye una solución nativa para esta tarea. Te encontrás frente a la opción de crear una solución personalizada desde cero, pero los tiempos son ajustados, y recién estás aprendiendo.

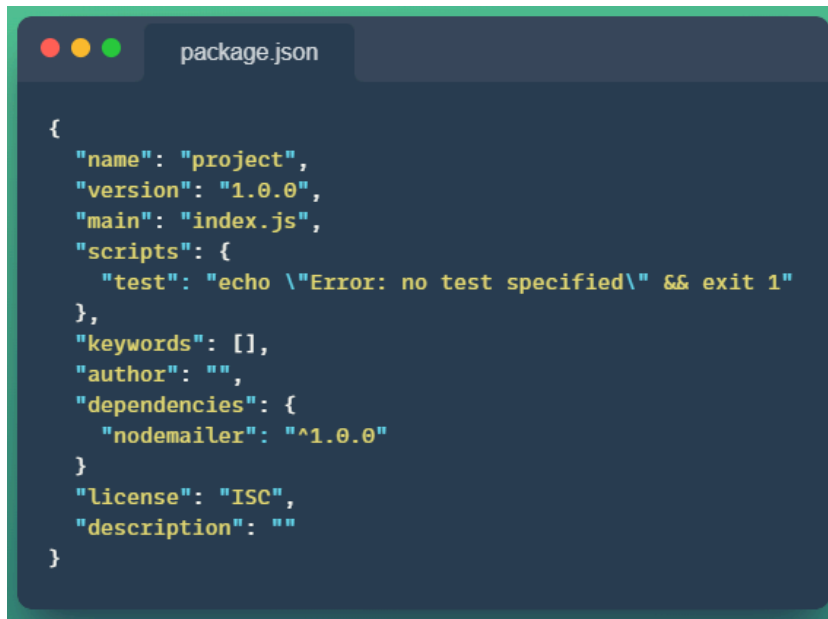
De pronto, recordás que NPM no solo es una herramienta para inicializar y configurar proyectos en Node.js, sino también una librería gigantesca de paquetes y código de terceros que resuelven necesidades específicas. Se te ocurre buscar si existe algún paquete que te permita enviar correos electrónicos de manera sencilla.

Afortunadamente, encontrás un paquete llamado **nodemailer**, diseñado precisamente para este propósito. Ahora, para poder utilizarlo en tu proyecto, tenés que aprender a instalar y gestionar dependencias con NPM. Veamos cómo hacerlo.

1. Para descargar e instalar un paquete, utilizamos el comando **npm install** seguido del nombre de la dependencia a instalar.



Este comando instala la dependencia como dependencia de “*producción*” lo que significa que será utilizada como parte necesaria de la ejecución final del proyecto y será agregada al objeto **dependencies** del archivo **package.json**.

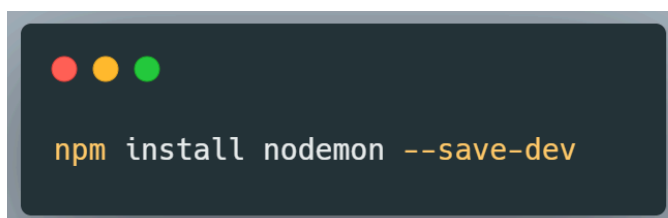


```
package.json

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

También podemos instalar dependencias de “desarrollo”, que serán aquellas utilizadas únicamente durante el proceso de desarrollo del proyecto. Normalmente son dependencias que ayudan o facilitan este proceso como pueden ser una librería llamada **eslint**, útil para estandarizar la escritura de código o **nodemon** para no reiniciar el proyecto manualmente frente a cada cambio en el código mientras desarrollamos.

Para instalar dependencias de desarrollo, usamos el mismo comando **npm install** y le agregamos el flag **--save-dev** o **-D**.



```
npm install nodemon --save-dev
```

Ahora tendremos una nueva propiedad en nuestro **package.json** que lista las dependencias de desarrollo:

```

package.json

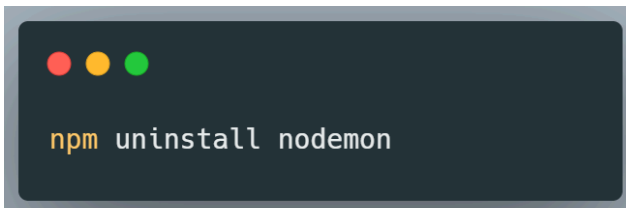
{
  "name": "project",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "dependencies": {
    "nodemailer": "^1.0.0"
  },
  "devDependencies": {
    "nodemon": "1.0.0"
  }
  "license": "ISC",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

- Es normal que estas dependencias reciban actualizaciones con mejoras o parches de seguridad y soluciones de errores. Para actualizar una dependencia debemos correr el comando **npm update** seguido del nombre de la dependencia, también podemos agregarle **@latest** o **@x.y.z** (número de versión) al nombre de la dependencia para instalar la última versión o una versión en específico.

```

npm update nodemon@latest
npm update nodemailer@3.5.0
    
```

- Para desinstalar una dependencia que ya no necesitamos, simplemente utilizamos **npm uninstall** seguido del nombre de la dependencia:



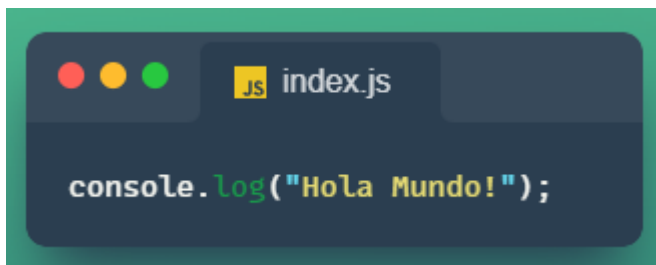
```
npm uninstall nodemon
```

De esta manera podemos gestionar de forma sencilla todas las dependencias externas que necesitemos en nuestros proyectos. Recuerda que **NPM** solo es una de las distintas opciones que existen en el mercado y cada una tiene su propia sintaxis aunque en el fondo todas sirven para lo mismo.

Creación de scripts

Otra de las ventajas que nos aporta el archivo **package.json** es permitirnos definir o preestablecer determinados **scripts** que nos permitirán ejecutar distintos procesos de nuestro código.

Supongamos que tenemos el siguiente programa:



```
index.js
console.log("Hola Mundo!");
```

Según lo visto anteriormente, sabemos que desde la terminal con el comando **node index.js** podemos ejecutar el código de este archivo sin problema, sin embargo, muchas veces los comandos de ejecución llevan variables de configuración lo que provoca que sean bastante más largos y difíciles de recordar, por eso podemos aprovechar nuestro archivo **package.json** para definir determinados comandos de ejecución:

Ahora, en la terminal podemos ejecutar el comando **npm run start** el cuál será interpretado por el programa como **node index.js**.

```

package.json

{
  "name": "project",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js"
  },
  "keywords": [],
  "author": "",
  "dependencies": {
    "nodemailer": "^1.0.0"
  },
  "devDependencies": {
    "nodemon": "1.0.0"
  },
  "license": "ISC",
  "description": ""
}
    
```

De esta manera estandarizamos nuestros comandos de ejecución y podemos crear tantos como sean necesarios bajo los nombres que deseemos.

Módulos

Node.js organiza el código en piezas reutilizables llamadas módulos. Estos permiten dividir la funcionalidad en diferentes archivos y aprovechar herramientas externas o integradas. En esta sección exploraremos los módulos internos, nativos y los módulos externos, comprendiendo cómo usarlos y sus diferencias.

¿Qué son los módulos?

Los módulos son bloques de código reutilizables y organizados que encapsulan la funcionalidad de una aplicación. Permiten dividir el código en partes más pequeñas y manejables, lo que facilita el mantenimiento, la reutilización y la colaboración en proyectos.



Node JS utiliza el sistema de módulos llamado **CommonJS** aunque también se puede utilizar el estándar **ESModules**. Ambos proporcionan una forma estándar de definir, importar y exportar módulos.

Cada archivo de JavaScript en Node.js se considera un módulo por defecto, lo que significa que el código dentro de ese archivo está contenido en ese ámbito y no afectará a otros módulos a menos que se exporten específicamente ciertos elementos.

Módulos Internos

Son todos los archivos de nuestro proyecto donde disponibilizamos o exportamos código que puede ser importado como módulo desde otro archivo.

Módulos Nativos

Node.js proporciona una amplia gama de módulos internos (core modules) que ofrecen funcionalidades listas para usar, como “http” para la creación de servidores web, “fs” para la manipulación de archivos, “path” para la manipulación de rutas de archivo, entre otros.

Módulos Externos

Además de los módulos nativos, Node JS cuenta con un vasto ecosistema de módulos externos disponibles en el repositorio npm (Node Package Manager) o el gestor de paquetes de nuestra preferencia.

Como aprendimos en la sección anterior, estos gestores contienen miles de módulos de código abierto que abarcan desde utilidades generales hasta frameworks y librerías especializadas.

Importando, exportando y creando Módulos.

Actualmente, en JavaScript existen dos estándares principales para importar y exportar código, cada uno con características y casos de uso particulares:

1. **CommonJS (CJS)**: Es el estándar más antiguo y ampliamente utilizado, con soporte nativo tanto en JavaScript como en Node.js. Ha sido el formato predeterminado para la gestión de módulos en Node.js durante muchos años.

CJS

Aquí un ejemplo de cómo exportar código con **CJS**:

```

// Exportando funciones y constantes
const add = (a, b) => a + b;
const subtract = (a, b) => a - b;

module.exports = { add, subtract };
    
```

Utilizamos la declaración **module.exports** donde seguido al símbolo de igualdad (=) le asignamos el código a exportar, en este caso será un objeto que contiene las funciones creadas. De ser una sola función simplemente exportaríamos el nombre de la misma.

Luego podemos importar este código de la siguiente manera:

```

// Importando con CommonJS
const math = require('./math');

console.log(math.add(3, 5)); // 8
console.log(math.subtract(10, 6)); // 4
    
```

En el archivo donde necesitamos implementar el código exportado, utilizaremos la sentencia **require('/ruta-al-archivo');** asignándose a una constante.

***Nota:** También podemos usar destructuring operator cuando importemos un objeto:
const { add, subtract } = require('./math');

2. **ES Modules (ESM):** Introducido en 2015 con la especificación ES6, este estándar, también conocido como EcmaScript Modules, representa una evolución en la forma de trabajar con módulos en JavaScript. ESM ha ganado popularidad por su sintaxis moderna y su integración nativa en navegadores y entornos modernos.

ESM

Aquí un ejemplo de cómo exportar mediante la sintaxis de **ESM**:

```

// Exportando funciones y constantes
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
    
```

A diferencia de CJS donde se exportan todas las funciones como un objeto en conjunto, ahora colocamos la palabra reservada **export** previo a la declaración de cada elemento a exportar.

Luego, al importar utilizamos la palabra reservada **import** seguido del destructuring de las funciones que deseamos utilizar y agregamos la palabra reservada **from** y la ruta donde se encuentra el módulo requerido.

```

// Importando con ESM modules
import { add, subtract } from './math.mjs';

console.log(add(3, 5)); // 8
console.log(subtract(10, 6)); // 4
    
```

Las principales diferencias entre **ES Modules** y **CommonJS** son:

- **Sintaxis:** ES modules utilizan una sintaxis basada en palabras clave como import y export para importar y exportar elementos. Por otro lado, CommonJS utiliza la asignación de objetos (module.exports y require) para exportar e importar elementos.
- **Comportamiento asíncronico:** los ES modules se cargan de forma asíncronica, lo que significa que las importaciones se resuelven dinámicamente en tiempo de ejecución. En cambio, CommonJS se carga de forma sincrónica, lo que significa que las importaciones se resuelven estáticamente en tiempo de compilación.

- **Ámbito:** Los ES modules tienen un ámbito propio por archivo, lo que significa que las variables y funciones declaradas dentro de un módulo no se filtran al ámbito global. En CommonJS, las variables y funciones declaradas en un módulo están disponibles en el ámbito global.
- **Exportación e importación estática:** ES modules solo permiten exportaciones e importaciones estáticas en la parte superior del archivo. No se pueden realizar exportaciones o importaciones condicionales o dentro de bloques de código. CommonJS permite exportaciones e importaciones dinámicas en cualquier parte del archivo.

***Nota:** Common JS es la sintaxis por defecto para el manejo de módulos en Node JS, en caso de querer utilizar **ES Modules**, es necesario especificar la propiedad **"type": "module"** en el archivo **package.json** de tu proyecto.

Dirname

En Node.js, `__dirname` es una variable global que representa la ruta absoluta del directorio que contiene el archivo actualmente en ejecución. Es particularmente útil para gestionar rutas y trabajar con archivos en proyectos donde las ubicaciones pueden variar entre entornos de desarrollo y producción.

¿Por qué usar `__dirname`?

Cuando trabajás con archivos y directorios en Node.js, necesitás proporcionar rutas exactas. Sin embargo, estas rutas pueden ser relativas al archivo que ejecuta el código. Esto puede generar problemas si cambias de directorio o despliegas tu aplicación en un servidor ya que dependiendo el entorno de ejecución, las referencias a ciertos archivos pueden verse modificadas.

Usar `__dirname` garantiza que las rutas sean absolutas, eliminando la dependencia de la ubicación actual desde donde se ejecuta el script.

Uso de `__dirname` en CJS.

Supongamos que tienes la siguiente estructura de directorios:

```
/proyecto
├─ index.js
├─ data
  └─ ejemplo.txt
```

Ahora queremos desde el archivo **index.js** acceder a la información dentro del archivo **ejemplo.txt**, para ello podemos utilizar el método llamado **readFileSync** del módulo nativo llamado **fs** (abreviación de file system) que nos brinda métodos para acceder, escribir, crear y eliminar archivos de la PC o servidor con Node JS.

En este caso **readFileSync** recibe como primer parámetro la ruta absoluta al archivo deseamos leer, donde lo más lógico sería lo siguiente:

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

// Indicamos la ruta en el primer parámetro

fs.readFile('c:/user-pc/node-project/data/ejemplo.txt', 'utf8',
  (err, data) => {
    if (err) {
      console.error('Error al leer el archivo:', err);
      return;
    }

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

Si bien esto podría funcionar, existen casos donde **por el contexto** donde se esté ejecutando el código (por ejemplo un servidor cuya estructura de carpetas podría no estar controlada por nosotros) la ruta indicada no apunte correctamente al archivo.

Por fortuna, gracias a la existencia de `__dirname` podemos saber en tiempo de ejecución la posición absoluta del archivo actual dentro del servidor, entonces utilizando esta variable global junto con `fs` y la ayuda de `path`, otro módulo nativo de Node que nos provee de métodos para trabajar con rutas de archivos de forma sencilla, podremos crear una referencia que nunca se vea corrompida, de la siguiente manera:

```
const path = require('path');
const fs = require('fs');

/* Obtenemos la ruta absoluta al archivo ejemplo.txt
   utilizando __dirname */

const filePath = path.join(__dirname, 'data', 'ejemplo.txt');

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

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

En el ejemplo anterior, `path.join()` combina el resultado de `__dirname` (supongamos que es algo como `C:/server/node-project/`) junto con el nombre de la carpeta y el nombre del archivo donde se encuentra `ejemplo.txt` para crear una referencia exacta dentro del contexto donde se está ejecutando el proyecto.

Analicemos este código paso a paso:

1. `__dirname` proporciona la ruta completa del directorio donde se encuentra `index.js`.
2. `path.join` combina `__dirname` con los subdirectorios y el nombre del archivo para formar una ruta absoluta.
3. `fs.readFile` usa esa ruta absoluta para acceder al archivo, asegurándose de que funcione sin importar desde dónde ejecutes el script.

Uso de `__dirname` en ESM.

En proyectos que usan ES Modules, `__dirname` no está disponible. En su lugar, puedes usar `import.meta.url` para obtener una ruta absoluta.

```
import { fileURLToPath } from 'url';
import path from 'path';

// Obtener el directorio actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

console.log('Ruta absoluta:', __dirname);
```

En esta ocasión aprovechamos el método `fileURLToPath` del módulo nativo `url`, el cual recibirá y formateará el valor de `import.meta.url` para devolvernos la posición actual de nuestro archivo y mediante el método `path.dirname()` tomaremos sólo la ruta (quitando el nombre del archivo) logrando el mismo resultado que en CJS.

Una vez obtuvimos la ruta absoluta podemos utilizarla del mismo modo que el ejemplo anterior:

```
import { fileURLToPath } from 'url';
import path from 'path';

// Obtener el directorio actual
const __filename = fileURLToPath(import.meta.url);
const __dirname = path.dirname(__filename);

console.log('Ruta absoluta:', __dirname);

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

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

Conclusión:

__dirname es una herramienta esencial en Node.js para trabajar con rutas absolutas de manera confiable. Junto con módulos como **path**, facilita la manipulación de rutas y archivos en tus aplicaciones, asegurando que funcionen sin importar el entorno o ubicación desde donde se ejecuten.

Process

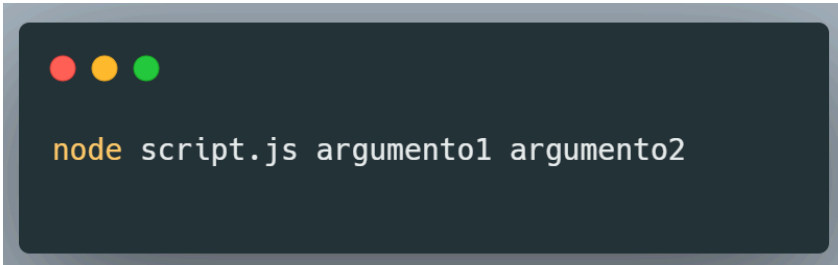
El módulo **process** es uno de los módulos nativos más importantes de Node.js, ya que permite interactuar con el proceso en ejecución. Una de sus propiedades clave es **process.argv**, un array que contiene los argumentos pasados al script desde la línea de comandos.

¿Qué es **process.argv**?

Es un array que incluye:


- La ruta absoluta del ejecutable de Node.js.
- La ruta del archivo ejecutado.
- Los argumentos adicionales proporcionados por el usuario.

Por ejemplo, si ejecutamos:



```
node script.js argumento1 argumento2
```

El contenido de **process.argv** será:



```
[  
  '/ruta/a/node',  
  '/ruta/a/script.js',  
  'argumento1',  
  'argumento2'  
]
```

De esta manera, podemos aprovechar **process.argv** para crear scripts interactivos que respondan a diferentes argumentos. Veamos un ejemplo simple:

```
const args = process.argv.slice(2);  
// Ignoramos los dos primeros elementos con slice  
  
if (args[0] === 'saludar') {  
  console.log(`¡Hola, ${args[1] || 'mundo'}!`);  
} else if (args[0] === 'despedir') {  
  console.log(`¡Adiós, ${args[1] || 'mundo'}!`);  
} else {  
  console.log('Comando no reconocido. Usa "saludar" o  
    "despedir".');  
}
```

Ahora en la terminal ingresamos:



```
node script.js saludar Matías
```

y obtendremos el siguiente resultado:



```
¡Hola, Matías!
```



En el ejemplo anterior, tenemos un programa que lee los 2 argumentos siguientes a las instrucciones **node script.js** que ejecutan el programa. En función de los valores ingresados retorna un resultado:

- Si el primer argumento es “saludar” o “despedir” enviará un mensaje de bienvenida o despedida, caso contrario informa que el comando no es el apropiado.
- Si existe un segundo argumento entonces el mensaje irá dirigido al nombre ingresado, caso contrario usará la palabra “Mundo” para formular el saludo.

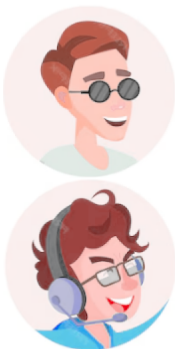
Consejos al utilizar **process.argv**

1. **Validar los argumentos:** Asegúrate de manejar casos en los que no se pasen suficientes argumentos o se usen comandos inválidos.
2. **Parámetros dinámicos:** Puedes usar los argumentos para realizar tareas específicas, como leer archivos, realizar cálculos, o manejar interacciones más complejas.

Ejercicio Práctico

Ejercicio 1 - ¡Construyendo la Base del Proyecto!

Matías y Sabrina te observan mientras revisas tus últimos desafíos resueltos. “Es momento de subir el nivel”, dice Matías, mientras Sabrina asiente.



“Queremos que configures un entorno de proyecto más profesional, algo que será clave si llegas a trabajar con nosotros en un futuro. Aquí tienes el siguiente reto”.

Misión:

- Crea un nuevo proyecto en Node Js mediante el comando `npm init -y` y configura un nuevo script dentro del archivo **package.json** que mediante la instrucción `npm run start` ejecute el código de nuestro archivo script.js de forma automática.

Asegúrate de que el entorno esté listo y funcionando correctamente. Este paso marcará el inicio de proyectos más complejos.

Ejercicio 2 - ¡Acciones dinámicas con Node.js!

Con el proyecto configurado, Matías lanza el siguiente desafío.



“El siguiente paso evalúa tu capacidad para manejar interacciones dinámicas en Node.js”, dice mientras sonríe. “Queremos que implementes un sistema simple para procesar comandos desde la terminal”.

Misión 2:

1. Si el comando es `npm run start GET`, imprime por consola el mensaje:
`Toma un dato`
2. Si el comando es `npm run start POST {data}`, imprime por consola el mensaje:
`Recibimos {data} satisfactoriamente`
3. Si el comando es `npm run start PUT {id}`, imprime por consola el mensaje:
`Modificamos el item con id: {id} satisfactoriamente`



4. Si el comando es `npm run start DELETE {id}`, imprime por consola el mensaje:

```
El item con el id: {id} se eliminó con éxito
```

Matías concluye: “Este desafío es clave para ver cómo manejas el flujo de datos y comandos, algo vital en cualquier proyecto backend”.

Materiales y Recursos Adicionales:

1. [Introducción a NPM](#) – Documentación oficial de Node.js para conocer más sobre el gestor de paquetes de Node.
2. [NPM Documentation](#) – Guía oficial sobre cómo trabajar con NPM y gestionar dependencias.
3. [Módulos en Node JS](#) - Documentación oficial sobre el manejo de módulos.
4. [Módulo process](#) - Leyendo argumentos desde la línea de comandos.

Preguntas para Reflexionar:

- ¿Cuáles son las diferencias clave entre CommonJS y ES Modules? ¿Cuál utilizarías en un proyecto moderno y por qué?
 - ¿Cuáles son los riesgos de no utilizar rutas absolutas en aplicaciones distribuidas?
 - ¿Qué ventajas encuentras en la automatización de comandos mediante scripts personalizados?
-



Próximos Pasos:

Manejo de Promesas: Exploraremos cómo manejar procesos asíncronos utilizando promesas y `async/await` en JavaScript.

Servidores Web: Aprenderemos sobre la comunicación web y cómo funcionan los servidores que dan vida a internet.

Patrones de Arquitectura: Conoceremos sobre los cimientos de los proyectos de programación.



Buenos Aires
aprende
Agencia de Políticas para el Futuro

BA Buenos
Aires
Ciudad