

Curso de Nodejs

Por Camilo Canclini

Este "curso" refleja mi propio aprendizaje con respecto a Nodejs. Ire recopilando todo lo que vaya aprendiendo y lo iré guardando en este repositorio. Me parece que es una forma interesante de estudiar y de demostrar como estudio. Sin mencionar que, por un lado, me sirve tener esta informacion guardada y documentada en un servicio online como github. Y, por otro lado, tambien soy consciente de que esto puede contruibuirle a alguna persona en el futuro. Por eso ten en cuenta que los siguientes "apuntes" provienen de alguien autodidacta que solo esta estudiando y que quiere compartir el conocimiento que obtenido.

Documentación Utilizada

Obviamente necesitamos tener algun tipo de documentación o trabajo previo sobre el cual basarnos a la hora de estudiar. En mi caso voy a recomendar, un video de YT, un roadmap y un libro. **Toda la información es gratuita y pública**

PDF [Node.js Notes for Professionals](#)

YT [Nodejs Curso Desde Cero, para principiantes | 1HS | FAZT](#)

YT [Nodejs Curso Práctico | 4HS | FAZT](#)

ROADMAP [Roadmap.sh | NodeJS](#)

CHATGPT [CHATGPT | OPENIA](#)

Conceptos Previos

En mi caso ya tengo conocimientos, redes informaticas, modelo OSI, modelo cliente-servidor, hardware, funcionamiento interno de la pc, etc. Para empezar a trabajar con Node se recomienda tener un minimo de conocimiento en estos temas ya que, al ser una tecnología que trabaja en el backend, todo estos conocimientos son fundamentales para entender la manera en la cual opera y como trabaja Node. Tampóco me centraré en explicar conceptos propios de javascript, ya qué, se supone que se debe tener conocimientos minimos en js para empezar a ver Node.

Introducción A NodeJS

Nodejs es un entorno de ejecución para Javascript y un manejador de eventos asíncronos

Vamos a empezar a desglozar esto.

- **Entorno de ejecución:** Significa que haremos uso de un script o programa para poder ejecutar Javascript, esto quiere decir que no haremos uso del navegador para correr el código, este correrá en nuestra máquina o servidor.
- **Manejador de eventos asíncronos:** Significa que este será capaz de escuchar, eventos del sistema y peticiones de usuarios. Los eventos son instancias en las que el estado de algún componente que está siendo "escuchado" cambia o "dispara" una alerta al servidor. Con asíncronos se refiere a que pueden ocurrir en cualquier momento de la ejecución, no existe un tiempo que marque cuando puede dispararse un evento.

¿Para que se usa Node ?

Node es utilizado para crear **aplicaciones web escalables**, esto quiere decir que estaremos creando un programa que pueda ser corrido desde nuestra propia pc o servidor y que además tendrá funcionalidades web. Con escalables se refiere a que la sintaxis y la forma en la que se construyen estas aplicaciones pueden cubrir necesidades básicas o más complejas, ya que, el entorno se ajusta dependiendo de la complejidad de sus funciones. Además otra ventaja es que, gracias a la asincronía, el sistema no se bloquea, por lo que es más fácil trabajar con este tipo de aplicaciones.

Requerimientos

Para empezar a trabajar con Node.js primero necesitamos descargarlo desde la página oficial. [Node.js.org](https://nodejs.org) La instalación realizará 2 cosas importantes que deberemos tener en cuenta, la primera es que al instalar Node también estaremos instalando npm, y la segunda es que se integrará el comando "node" a nuestro PATH.

¿Qué es NPM?

NPM significa Node Package Manager o en español Gestor de Paquetes de Node. Esta es una herramienta que permite la instalación de paquetes para nuestro entorno de ejecución. Los paquetes son, un conjunto de módulos o archivos javascript los cuales cumplen funcionalidades específicas, y pueden ser llamados desde nuestra aplicación de manera sencilla y práctica. Frameworks como React pueden ser integrados a través de esta herramienta.

¿Qué es PATH?

El Path es el listado de las rutas a programas que pueden ser llamados desde nuestra terminal, o sea, en este caso, el programa o script "node" se utilizará desde la terminal.

Hola Mundo en Node.js

Recomiendo que para la próxima lección guardemos el siguiente código en un archivo holaMundoNode.js

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

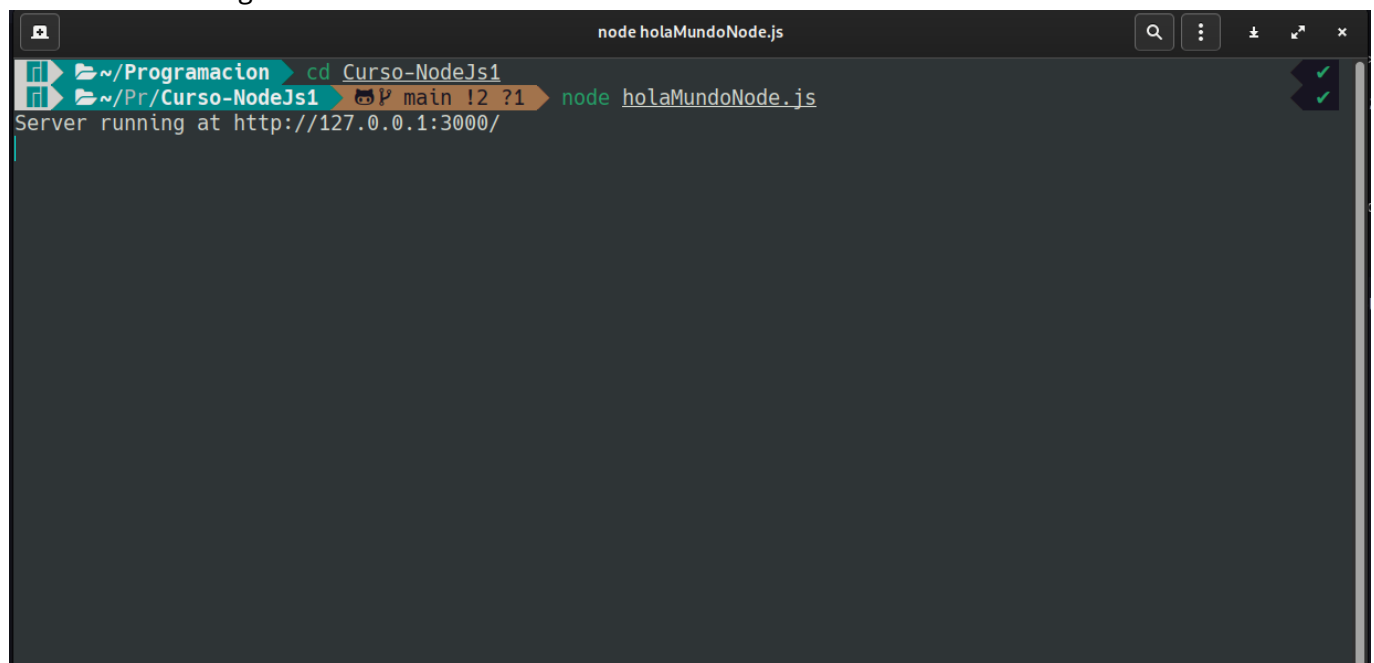
const hostname = '127.0.0.1';
const port = 3000;
```

```
const server = http.createServer((req, res) => {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World');  
});  
  
server.listen(port, hostname, () => {  
  `Server running at http://${hostname}:${port}/`;  
});
```

Desde la terminal nos dirigimos a la carpeta donde guardamos el script anterior (`cd Curso-Nodejs`) y ejecutamos el comando `node holaMundoNode.js`

Luego nos dirigimos a la ruta que nos indica la aplicación en mi caso, `http://127.0.0.1:3000/`.

Deberíamos ver algo como esto...

A screenshot of a terminal window titled 'node holaMundoNode.js'. The terminal shows the command 'cd Curso-NodeJs1' being executed in the directory '~ / Programacion'. Below that, the command 'node holaMundoNode.js' is executed in the directory '~ / Pr / Curso-NodeJs1', resulting in the output 'Server running at http://127.0.0.1:3000/'. The terminal interface includes a dark background with light-colored text and a sidebar on the left showing file explorer icons.



Hello World

Desglozando el Hola Mundo

En este caso se hace uso del modulo `http`, es se utiliza para montar el servidor en nuestra pc y es el que permite responder a los usuarios con información. Con respecto a los modulos hay algo que tenemos que remarcar y es el tipo de modulo al que se esta haciendo referencia, en este caso, el modulo `http` es un modulo que ya viene instalado con Node por lo que no es necesario hacer ningun llamado "especial" o descargarlo de npm por ejemplo. Este tipo de modulos se los conoce como **Modulos Core**, ya que vienen preinstalados. (Un poco más adelante veremos sobre módulos)

Una vez importado en la linea uno, se guardan en variables constantes los datos para configurar el servidor (`const hostname = '127.0.0.1';` y `const port = 3000;`)

Ahora bien, a continuación podemos ver uno de los conceptos mas importantes que se mencionaron anteriormente. Vease el siguiente fragmento...

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

Lo que esta ocurriendo aquí es que se esta guardando el resultado de un metodo del objeto `http` que importamos antes. Fijese como, por parametros estamos pasando una **función flecha**. Esto es fundamental, ya que, permite que el código siga ejecutandose a pesar de que el proceso dentro del bloque de la función no haya terminado.

Esto es lo que hablamos antes, famosa **asincronía** de tareas o procesos, evita que el programa se bloquee y no pueda continuar. Si bien es un ejemplo sencillo, gracias a el podemos entender muchos otros conceptos y modulos.

Despues vemos como por parametros le pasamos a la función flecha 2 variables, que despues se transforman en objetos:

- `req` : es el objeto que permite manejar los mensajes entrantes (En este caso no se usa).
- `res` : es el objeto que permite manejar la respuesta que da el servidor al cliente que hace una petición. Este es el importante ya que setea el tipo de respuesta, el codigo de esta HTTP, y el mensaje (en este caso, hello world).

Para mas informacion, visita [Nodejs.Org](https://nodejs.org/en/docs/api/http) | [http.methods](https://nodejs.org/en/docs/api/http)

Por otro lado tenemos...

```
server.listen(port, hostname, () => {  
  `Server running at http://${hostname}:${port}/`;  
});
```

Este metodo `server.listen()` es el que setea donde se esta ejecutando la aplicación web, en que dirección y que puerto. Y ademas nos permite realizar operaciones mientras que se incia. Ya qué, al igual que el anterior, esté acepta una función asíncrona, que en este caso arroja un mensaje por terminal que nos indica en que dirección se encuentra corriendo el servidor.

Modularización

La modularización es un concepto de la programación que hace referencia al hecho de dividir el código en distintos fragmentos y cada uno realice una tarea específica. Las ventajas de esta metodología son:

- Mejor organización
- Mejor mantenimiento
- Mejor rendimiento
- Reutilización de codigo

Hasta ahora vimos la utilización del modulo `http`, el cual es un fragmento de codigo que permite gestionar las peticiones por el protocolo http. Esto nos facilita esa tarea en concreto y hace que no tengamos que preocuparnos por crear nosotros esa funcionalidad.

Otra ventaja, es que si pensamos los modulos como piezas que encastran entre si, podemos decir que a la hora de construir un proyecto seremos capaces de elegir que piezas utilizar en nuestro proyecto y cuales no.

El objeto `module` En NodeJS

El objeto `module` es un objeto que se comparte entre modulos, este guarda atributos y metodos que se mantienen durante la ejecución del programa.

Veamos como se compone:

```
Module {
  id: '.',
  path: '/home/camilocanclini/Programacion/Curso-NodeJs1',
  exports: {},
  filename: '/home/camilocanclini/Programacion/Curso-NodeJs1/holaMundoNode.js',
  loaded: false,
  children: [],
  paths: [
    '/home/camilocanclini/Programacion/Curso-NodeJs1/node_modules',
    '/home/camilocanclini/Programacion/node_modules',
    '/home/camilocanclini/node_modules',
    '/home/node_modules',
    '/node_modules'
  ]
}
```

Entre las propiedades mas importantes que guarda se encuentra `exports`, que guarda un objeto. A continuación veremos para que se utiliza...

Tipos de módulos en JS

CommonJS modules

Es la forma original en la cual el modulo se preparaba para ser importado, es el que viene por defecto integrado en el lenguaje de JS.

Vease el siguiente ejemplo:

Supongamos que el siguiente modulo es un archivo llamado `foo.js`

```
module.exports.add = function(a, b) {
  return a + b;
}

module.exports.subtract = function(a, b) {
  return a - b;
}
```

Lo que esta ocurriendo aqui es que se esta haciendo uso del objeto global `module`. Debido a la característica ya mencionada este permite que se "comparta" su propiedad `exports` entre modulos. Por ejemplo, el siguiente modulo, que denominaremos `main.js`

```
const {add, subtract} = require('./foo')

add(5, 5) // 10
subtract(10, 5) // 5
```

Resaltemos algo importante, notece como en la declaración de `const {add, subtract} = require('./foo')` las constantes estan encerradas entre `{}`, esto es una funcionalidad que se agregó en ES6 llamada "destructuring assignment".

Lo que permite la "asignación de deestructuración" es acceder directamente a las propiedades o metodos de una objeto (en este caso) ignorando todas las que no sean especificadas. En el código dado anteriormente el objeto es `module`, el cual, *solo* contiene las 2 funciones que importamos desde `./foo.js`, entonces si yo al momento de declarar las constantes las declaro entre `{}` lo que va a ocurrir es que solo se hará uso de esas 2 funciones. En el caso de que el objeto `module` contenga mas funciones ademas de `add()` y `subtract()`, estas seran "**ignoradas**" y solo se importaran las que especifiquemos.

Ademas notece una última cosa, fijese que al momento de pasar por parametro en `require()` la ubicación del modulo que queremos importar, este comienza con un `./`

Esto significa que estamos buscando el modulo **desde la misma carpeta** que el "archivo llamador", en este caso seria desde la carpeta o directorio donde se encuentra `main.js`

EMACASript modules (ES Modules)

Por otro lado tenemos los ES Modules, que son modulos de javascript que se encuentran estandarizados. Esto quiere decir que estos tienen una estructura o sintaxis diferentes que los hace poseer mejores ventajas que los CommonJS Modules. Vease el anterior codigo pero esta vez como la metodologia de un ES Module...

```
export function add(a, b) {
    return a + b;
}

export function subtract(a, b) {
    return a - b;
}
```

Fijese que ahora la sintaxis es mucho mas limpia y agradable a la vista. Algo que tenemos que aclarar es que el codigo de arriba **NO** lo guardaremos como `foo.js` **SINO** como `foo.mjs`.

Esto permite que Node identifique que el modulo se trata del tipo estandarizado. Pero ahora bien, como seria el archivo `main.js`?, vease el siguiente codigo...

```
import {add, subtract} from './foo.mjs'

add(5, 5) // 10
subtract(10, 5) // 5
```

Vease que la forma de importar tambien cambia.

Diferencias (import and require)

- Los `import` solo pueden ser llamados desde el principio del archivo (`main.js`), mientras que los `require()` pueden ser llamados en cualquier momento, por ejemplo.

```
if(user.length > 0){
  const userDetails = require('./userDetails.js');
  // Do something ..
}
```

- Los `import` son **ASINCRONOS** y los `require()` son **SINCRONOS**, lo que quiere decir que, los `require()` esperan a que se terminen de cargar todas las funciones para poder continuar la ejecución. Esto puede perjudicar al rendimiento en grandes aplicaciones.

El objeto `global` En NodeJS

El objeto `global`, es similar al objeto `window`, los 2 almacenan los objetos y metodos que comunmente usamos cuando trabajamos con JS, por ejemplo:

- `console`
- `setInterval()`
- `setTimeout()`

La diferencia es que `window` se usa cuando js se esta ejecutando en el navegador y el otro se utiliza en Node.

Los tipos de variables `var` en Node

Algo importante a tener en cuenta cuando hablamos de módulos es que el comportamiento que toman las variables del tipo `var` cambia, ya que en node, su **scope** es el modulo desde donde se declaran. A diferencia del navegador que se comparte entre archivos porque se almacena en el objeto `window`.

NPM (Node Package Manager)

Ahora vamos a ver en mayor profundidad NPM. Como digimos anteriormente, este sirve para instalar paquetes que son consumidos por nuestra aplicación para realizar tareas especificas.

Preparando proyecto

Para empezar a utilizar NPM se recomienda que empecemos con el siguiente comando.

```
npm init
```

Este comando lo que hará será hacernos una serie de preguntas relacionadas con los metadatos del proyecto entero, posteriormente esto creará un primer archivo para mantener esta configuración, llamado `package.json`.

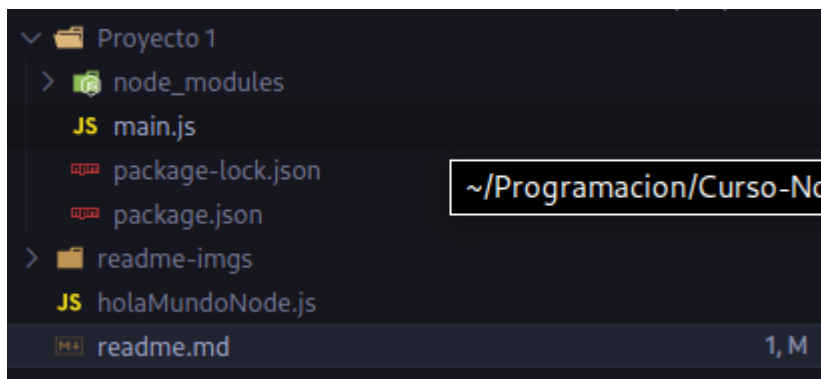
Como instalar paquetes

Una vez instalado nodejs y npm, tan solo bastaría con abrir la terminal y navegar hasta la carpeta del proyecto para escribir el siguiente comando

```
npm install <NombreDelPaquete>
```

Este comando instalará el paquete indicado, creando una carpeta en la raíz de nuestro proyecto llamada `node_modules`. Aquí se almacenarán todos los paquetes.

package.json y package-lock.json



Estos 2 se crean al momento de empezar a instalar paquetes o dependencias, como dijimos `package.json` es el archivo donde se guarda la configuración de la carpeta raíz, en mi caso Proyecto1. Mientras que el `package-lock.json` es el archivo donde se va a guardar el listado de módulos descargados, así como un historial de versiones de los mismos.

Como instalar dependencias

Si nosotros bajamos un proyecto y necesitamos instalar *todos* los paquetes necesarios para que este funcione, tan solo bastaría, ir a la terminal, dirigirnos al directorio raíz del mismo proyecto y escribir el siguiente comando:

```
npm install
```

Esto lo que hará será descargar todos los paquetes listados en el directorio (Que se supone que existe)

Como actualizar paquetes

Para un actualizar paquete usamos

```
npm update <NombreDelPaquete>
```

Para actualizar **TODOS** los paquetes

```
npm update
```

Como importar paquetes

```
require(<NombreDelPaquete>)
```

Con el metodo `require()` podemos hacer uso de los paquetes instalados en el proyecto. **No es necesario especificar ninguna ruta, solo indicar el nombre.**

Al integrar estos paquetes estos pasan a convertirse en **dependencias** que nuestra aplicación necesita para funcionar.

NPM Tasks

Si miramos el archivo `package.json` deberiamos ver algo parecido a lo siguiente:

```
{
  "name": "proyecto-1",
  "version": "1.0.0",
  "description": "Este es el primer proyecto del curso",
  "main": "main.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "curso"
  ],
  "author": "Camilo Canclini Stephano",
  "license": "ISC",
  "dependencies": {
    "colors": "^1.4.0"
  }
}
```

Si nos fijamos en la parte de `scripts` podremos ver que se esta almacenando un objeto (keys y values). Este objeto, en mi caso, se encuentra guardando la siguiente entrada: `"test": "echo \"Error: ..."`.

Gracias a la estructura que brinda este archivo para el apartado de "**scripts**" es que somos capaces de guardar diferentes scripts, y **personalizados**. Esto nos permite generar una serie de comandos predefinidos que pueden ser leídos y ejecutados a través de npm con la finalidad de poder ejecutar ciertas acciones en nuestro paquete o proyecto que estemos creando, como por ejemplo: configurar e iniciar el servidor, crear archivos iniciales, ejecutar testeos, entre otras cosas.

Ejecutar Scripts con NPM

Con el siguiente comando seremos capaces de ejecutar los scripts que estén definidos en nuestro **package.json**.

```
npm run <NombreDelScript>
```

NPX

npx es un comando que se integró con npm para poder realizar la ejecución de comandos basándose en el **contexto de los paquetes**. Con contexto nos referimos a la **forma** en la que **tienen** que ser **accedidos** estos paquetes para funcionar.

Hay veces que los paquetes requieren estar instalados de manera **global** en el equipo, pueden requerir **privilegios de administrador** para ejecutar cierta función, o bien pueden necesitar ser ejecutados mediante la aplicación que estemos desarrollando directamente. NPX surge para solucionar todo esto.

NPX lo que hace es realizar todas las operaciones necesarias en un segundo plano para poder cumplir con el comando que queramos ejecutar. Un ejemplo muy sencillo podría ser el siguiente.

```
npx create-react-app prueba
```

create-react-app es un comando propio del framework de **React** que lo que hace es realizar las operaciones internas de la librería para crear la estructura de una app con React. Obviamente para que funcione requiero tener instalado React en mi proyecto, pero si uso npx, esto no es necesario. Al ejecutarlo nos preguntará si queremos instalar React, y si seleccionamos que sí, entonces npx lo instalará de forma temporal y lo posicionará basándose en el contexto de nuestro proyecto. Lo "instalará" manera local, en otras palabras.

Podemos concluir con que NPX es un comando que permite saltarse ciertos requerimientos como, por ejemplo, tener que declarar un comando en específico en el **package.json** para ejecutar cierto módulo.

Errors Handling

También conocidos como manejadores de errores, son las formas en las cuales se tratan los errores. Como dijimos antes, nuestra aplicación podrá ser accedida por muchos usuarios, y si esta escala requerirá de más archivos y funcionalidades.

Este tipo de aplicaciones no deberían ser detenidas por un simple error de código, esta tiene que poder responder siempre, aunque sea arrojando un mensaje de error. Se tiene que evitar detener la ejecución e impedir que los usuarios que disparen algún error sigan repercutiendo en el resto del sistema.

Tipos de errores

Existen 2 grandes tipos de errores:

Errores de Programador

Estos son el tipo de errores que se encuentran en el código y que dependen exclusivamente de la forma en la cual se programó la aplicación. Estos pueden ser manejados simplemente optimizando y depurando el código.

Errores Operacionales

Estos son los más complejos, ya que, dependen de **factores externos** a la programación de la aplicación. Son inesperados y se disparan a gracias a las operaciones que realizan los usuarios con nuestra aplicación o también debido a la forma en la que las distintas partes de nuestra aplicación se comunican entre sí.

Por ejemplo, un usuario podría estar intentando leer un archivo que se encuentra vacío, podría ingresar información que puede generar vulnerabilidades en el sistema, o un script podría ejecutar alguna operación sobre un archivo que no existe aún.

El objeto `Error`

En Node.js existe el objeto `Error`, el cual permite crear instancias para "lanzar" errores en la aplicación.

```
new Error("Aquí va el mensaje personalizado")
```

Algunos de sus propiedades son:

- `name`: Guarda el nombre del error
- `message`: Guarda un texto que describa el error que ocurrió
- `stack`: Guarda el camino que recorrió el error hasta ser arrojado, esto es sumamente útil, porque permite analizar función por función y bloque por bloque lo que se estaba ejecutando al momento de dispararse el error, esto también es conocido como **stack trace**.

Formas de manejar Errores

A continuación vamos a presentar algunas formas de manejar errores:

Bloque Try and Catch

Los bloques `try`, `catch` y `finally` son sentencias muy parecidas a los `if` de toda la vida, la diferencia radica en que son especiales para errores y evitan que se termine la ejecución del proceso actual.

Veamos un ejemplo:

```
var fs = require('fs')

try {
  const data = fs.readFileSync('/Users/user/file.txt')
} catch (err) {
  err)
}
finally{
  'Finally will execute every time')
}
```

Vamos por partes:

- **try**: va a ejecutar un fragmento del código, si este arroja cualquier tipo de error, **no detiene la ejecución** y en cambio "le pasa" el error a la sentencia **catch**, si no ocurre un error no arroja nada. En este caso hace uso del modulo **fs** para leer un archivo(la veremos mas adelante). Como es evidente la lectura puede fallar por causas externas a la aplicación, por esa razon se coloca dentro de **try** para ser "evaluada".
- **catch (err)**: Esta sentencia recibe el error arrojado por la sentencia **try**, y realiza algun tipo de operación en consecuencia del mismo. En este caso devuelve el error por consola (Pero no se detiene la aplicación).
- **finally**: Este bloque se ejecutará **independientemente del resultado anterior**, no es necesario agregarlo.

Recordemos que en este caso tenemos que evitar a toda costa que la aplicación se bloquee, ya que, necesitamos que, este disponible siempre para los usuarios, y que opere de manera autónoma.

Programación Asíncrona

Funciones Callbacks

Las funciones callbacks son aquellas que pueden ser pasadas como argumentos de otra función. En este caso, las usaremos para realizar operaciones si aparece un error.

Generalmente estas se pasan como argumento *final* de la función principal, estas son llamadas cuando la funcion desde la que se llama a la callback espera necesita un resultado, o cuando se dispara un error. Actuan como manejadores de errores porque permiten ejecutar operaciones si existe un error. Por ejemplo:

```
function operacionLargaConError(callback) {
  setTimeout(() => {
    const error = Math.random() < 0.5;
    if(error){
      callback(new Error('Ocurrio un error!'));
    }else{
      callback();
    }
  });
}
```

```
    }  
  }, 1000);  
}  
  
function manejarError(error) {  
  if (error) {  
    console.error(error);  
  } else {  
    'La operacion finalizo correctamente';  
  }  
}  
  
operacionLargaConError(manejarError);
```

En el código anterior se le pasa a la función `operacionLargaConError()` como argumento la función `manejarError(error)` que a su vez también recibe un parámetro que es el error disparado, en la función anterior.

Esta forma de operación permite separar la lógica principal del manejo de errores. Haciendo que el código sea modular y más legible a la vista.

Las callbacks son utilizados para seguir con la ejecución del código y tener que esperar al resultado de una función o método que tiene que devolver algo. Ya que, en NodeJS y JS la mayoría de procesos son asíncronos, necesitamos realizar varios procesos en simultáneo y que el código siga ejecución.

main.js

```
var1 = funcion1(arg1,arg2,...,argN,callback1( argC1, argC2, ..., argCN){  
  operacion1();  
  operacion2();  
  ...  
  operacionN();  
});
```

```
var2 = funcion2(arg1,arg2,...,argN,callback2( argC1, argC2, ..., argCN){  
  operacion1();  
  operacion2();  
  ...  
  operacionN();  
});
```

Como podemos ver en el dibujo anterior al momento de ejecutar el `main.js`, se guarda en `var1` el resultado de una función, y esta tiene un callback que va a ejecutar operaciones en segundo plano (sector rojo).

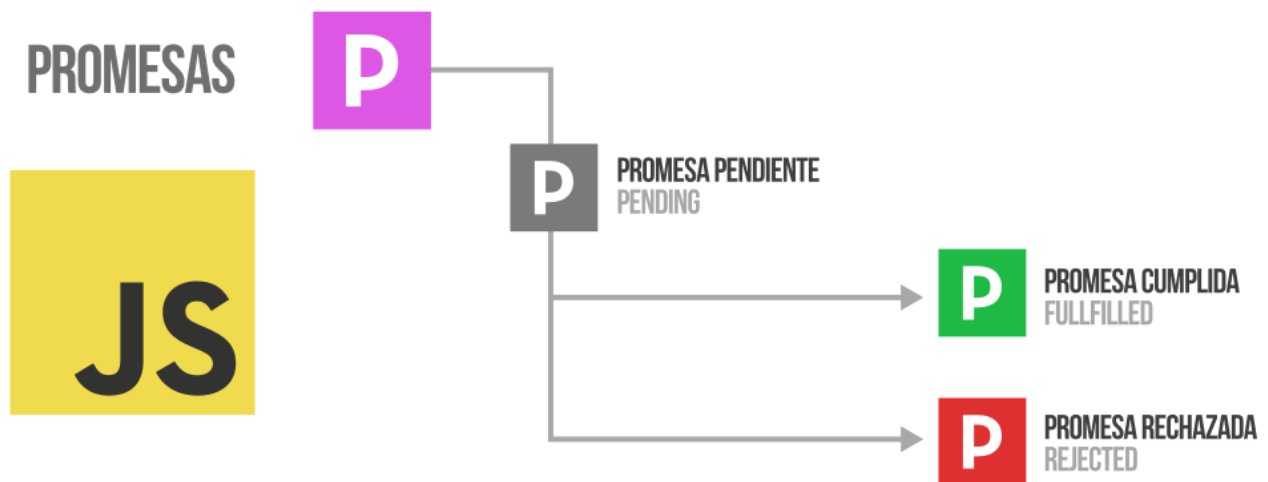
Por otro lado, mientras el `callback1` ejecuta sus operaciones, se estará definiendo, `var2`, la cual también depende del resultado de una función, y esta a su vez también posee un callback que estará ejecutando operaciones en segundo plano.

Al final las 2 variables se definirán casi a la par. Antes hubiésemos tenido que esperar a que se definiera completamente `var1` para poder empezar a definir `var2`, con las callbacks esto ya no representa un problema.

Promesas - Promises

Una promesa es un objeto que representa la terminación de una operación asíncrona, se utilizan para manejar tareas asíncronas como las que venimos viendo, a diferencia de las callbacks estas son mas sencilla de entender y de visualizar. La promesa tiene 3 estados:

- pending (pendiente)
- fulfilled (cumplida)
- rejected (fallida)



Una vez que cambia su estado a fulfilled o rejected no se puede volver para atras.

El objeto promesa tiene dos métodos que se usan para manejar los resultados de una promesa:

- `.then`: este método recibe una función que se ejecuta si la promesa se cumple.
- `.catch`: este método recibe una función que se ejecuta si la promesa falla.

Veamos un ejemplo:

```
const promesa = new Promise((resolve, reject) => {
  setTimeout(() => {
    if (Math.random() < 0.5) {
      resolve("Todo bien");
    } else {
      reject(new Error("Algo salió mal"));
    }
  }, 1000);
});

promesa.then(response => response)
promesa.catch(error => console.error(error));
```

Como podemos apreciar, primero se instancia el objeto promesa con `new Promise((resolve, reject) => {})`; la función que ejecuta la promesa se llama `executor` y sus argumento son los objetos: `resolve` y `reject`. Estos se asocian con los 2 metodos que mencionamos arriba.

Luego se empieza una función flecha y adentro un `setTimeout` en el cual vuelven a aparecer `resolve` y `reject`. Aquí es donde se hacen las operaciones para resolver la promesa. Recordemos que la promesa tiene solo 2 resultados posibles, `resolve(response)` para cuando todo sale bien y se pasa algún valor como parámetro, y `reject(Error)` cuando ocurrió algo que pueda ser considerado un error, en este último se pasa por parámetro un objeto `Error`.

Dependiendo de lo ocurrido en el bloque del `setTimeout` se ejecutan los métodos `.then` y `.catch`, para cuando todo sale bien y para cuando ocurrió un error respectivamente.

Por último, también existe el método `.finally`, que al igual que con `try` y `catch`, se ejecuta, independientemente de la resolución de la promesa. Aquí un ejemplo:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('La operación ha sido completada');
  }, 1000);
});

promise
  .then((result) => {
    result;
  })
  .catch((error) => {
    error;
  })
  .finally(() => {
    'Se ha completado la promesa INDEPENDIENTEMENTE del resultado';
  });
```

Ventajas de usar promesas

- A la hora de manejar errores el código se vuelve más legible. Evitamos los "callbacks hell"
- Se pueden concatenar varias promesas, véase el siguiente ejemplo:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Resultado de la operación 1');
  }, 1000);
});

let promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Resultado de la operación 2');
  }, 2000);
});

promise1.then((result) => {
  result;
});
```

```
    return promise2;
  }).then((result) => {
    result);
});
```

Podemos ver como se resuelve la segunda promesa solo si antes se resolvió la primera.

- Realizar varias operaciones asincronas en paralelo, vease el siguiente ejemplo:

```
let promise1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Resultado de la operación 1');
  }, 1000);
});

let promise2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('Resultado de la operación 2');
  }, 2000);
});

Promise.all([promise1, promise2]).then((results) => {
  results);
});
```

Aquí hacemos uso del metodo `.all` del objeto `Promise`, este recibe como parametro un arreglo con las promesas y si todas se realizaron correctamente, entonces se continua con la ejecución con `.then`. Recordemos que esto no detiene la aplicación en ningun momento, ya que son procesos asíncronos.

- Concatenar varios `.then`, Utilización del metodo `fetch()`

```
fetch('https://api.example.com/data')
  .then((response) => {
    return response.json();
  })
  .then((data) => {
    data);
  });
```

En este ejemplo, utilizamos la función `fetch` para realizar una solicitud HTTP asíncrona a una URL específica. El resultado de la solicitud se devuelve como un objeto llamado `response`, y a su vez el metodo devuelve una promesa, por lo que podemos hacer uso de los metodos propios de las promesas.

Como podemos ver, la lógica detras de esto esta en que, al resolver el primer `.then` se devuelve un promesa, y esa es sobre la que trabaja el segundo `.then`, con la promesa resuelta del primero.

Async y Await

La palabra reservada `async` se utiliza para definir funciones asíncronas, estas son, bloques de código que se ejecutan en "segundo plano", que no traban la ejecución del programa entero, y que **siempre** retornan una promesa. Al retornar una promesa se pueden aprovechar todas las ventajas mencionadas anteriormente. He aquí un ejemplo:

```
async function getData() {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Datos obtenidos');
    }, 1000);
  });
}

getData().then((result) => {
  result);
});

'Esto se ejecuta primero')
```

En este ejemplo podemos ver que la palabra `async` se pone antes de `function`, lo que convierte al bloque de código en una función asíncrona. Además vemos que cuando se llama a la función, inmediatamente se accede a la promesa (una vez que se resuelva en 1000ms) con `.then`. Por último, fijémonos que la última línea es un `)`, esta línea se ejecutará **antes** que la función, por más que la función haya sido llamada antes. Como sabemos que la función devuelve una promesa, podemos ejecutar todo el código que queramos sin necesidad de esperar a que la función entregue un resultado.

Por otro lado tenemos al `await`. Este lo que hace es **detener la ejecución de la función asíncrona** hasta que se resuelva una promesa, vease el siguiente ejemplo:

```
async function getData() {
  let result = await fetch('https://api.example.com/data');
  let json = await result.json();
  return json;
}

getData().then((data) => {
  data);
});
```

Miremos que el `await`, en este caso, va a esperar a que el método `fetch` retorne una promesa como respuesta, hasta que esto no ocurra no se realizará la línea siguiente. Vuelvo a recalcar, solo detiene la ejecución **de la función**, lo que quiere decir que todo el código que se encuentre afuera de la misma seguirá su ejecución normalmente.

El `await` solo está habilitado en funciones asíncronas.

setImmediate y process.nextTick

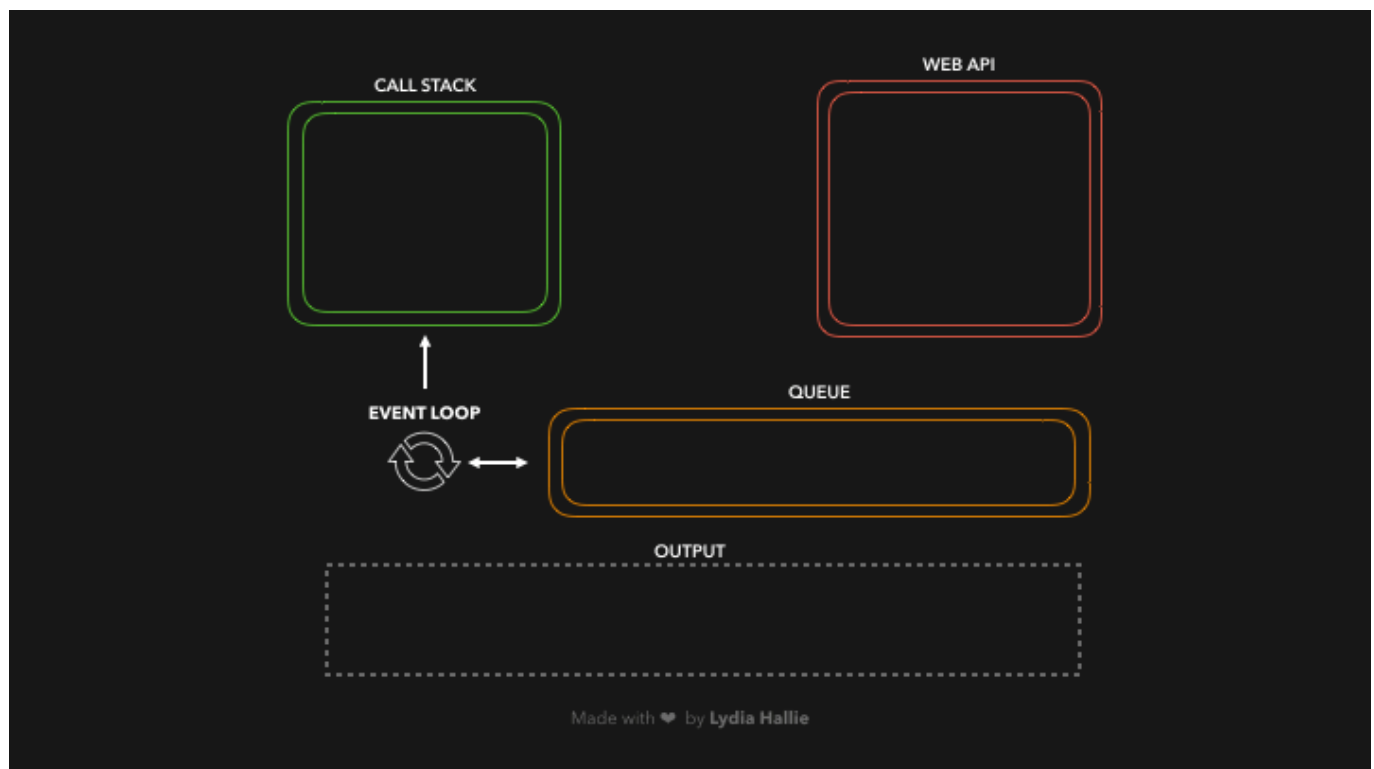
Para entender el funcionamiento de `setImmediate()` y `process.nextTick()`, primero necesitamos conocer bien que es el event loop en javascript.

Event Loop de JS

El event loop es la forma en la cual Javascript ejecuta y da orden a las tareas asíncronas. Además este consta de varias "fases" o "ciclos" que se repiten de forma continua, cada una de las cuales se encarga de procesar un conjunto específico de tareas. Algunas de estas fases son:

- Fase de Entrada: en esta fase, el event loop revisa las colas de eventos para ver si hay algún evento nuevo para procesar, como por ejemplo, un evento de click en un botón.
- Fase de Ejecución: en esta fase, el event loop ejecuta las tareas correspondientes al evento detectado en la fase anterior. Por ejemplo, ejecutar la función asociada a un evento de click en un botón.
- Fase de Salida: en esta fase, el event loop limpia cualquier información o estado de las tareas ejecutadas en la fase anterior, para estar preparado para el próximo ciclo del event loop.

JavaScript es un lenguaje single-threaded, esto significa que solo puede ejecutar una tarea a la vez. Sin embargo, al usar el event loop, se logra simular la concurrencia permitiendo ejecutar varias tareas al mismo tiempo sin bloquear el hilo principal de ejecución, esto es el famoso 'paralelismo'.



- Call Stack: Es una estructura de datos que almacena las funciones y las variables en ejecución. Cada vez que se llama a una función, se añade una entrada al call stack, y cada vez que se regresa de una función, se elimina una entrada del call stack. El call stack es el lugar donde **se lleva a cabo la ejecución de las instrucciones del programa**.
- Event queue: es una estructura de datos que contiene los eventos y tareas que deben ser procesadas por el event loop. Por ejemplo, un evento de click en un botón o una petición HTTP. Cada vez que un

evento ocurre, se agrega a la cola de eventos. El event loop revisa esta cola de forma continua y ejecuta las tareas correspondientes.

- Web API: Son las funciones y objetos proporcionados por el navegador (como el objeto `setTimeout`, `setInterval`, el objeto `XMLHttpRequest`). Cada vez que una función de la Web API es llamada, se ejecuta de manera asíncrona y una vez completada, agrega una tarea a la cola de eventos, para ser ejecutada por el event loop.

Ahora bien, `setImmediate()` y `process.nextTick()` son dos funciones que se utilizan para programar la ejecución de tareas en el futuro en Node.js. Sin embargo, existen algunas diferencias importantes entre ellas:

- `setImmediate()`: La función `setImmediate()` programa una tarea para ser ejecutada inmediatamente después de que todas las operaciones pendientes en el event loop actual hayan sido completadas. Es decir, `setImmediate()` se ejecuta en el siguiente ciclo del bucle de eventos.
- `process.nextTick()`: La función `process.nextTick()` programa una tarea para ser ejecutada en el siguiente ciclo del bucle de eventos, antes de que se ejecuten cualquier otra tarea programada con `setImmediate()`. **Es decir, `process.nextTick()` se ejecuta antes de que `setImmediate()` se ejecute.**

Un ejemplo sería:

```
"Ejecutando tarea 1");

setImmediate(() => {
  "Ejecutando tarea 2 (setImmediate)");
});

process.nextTick(() => {
  "Ejecutando tarea 3 (process.nextTick)");
});

"Ejecutando tarea 4");

//SALIDA

// Ejecutando tarea 1
// Ejecutando tarea 4
// Ejecutando tarea 3 (process.nextTick)
// Ejecutando tarea 2 (setImmediate)
```

Conceptos Importantes antes de ver los Core Modules

File Descriptors y El objeto `FileHandle`

Antes de empezar a explicar los metodos del módulo necesitamos aclarar los algunos conceptos. Al momento de interactuar con archivos se recurrirá a los binarios (programas) que ofrece el sistema

operativo donde nos encontramos, ya que en realidad, al momento de acceder a los archivos, necesitamos comunicarnos con el SO.

Cosas a tener en cuenta:

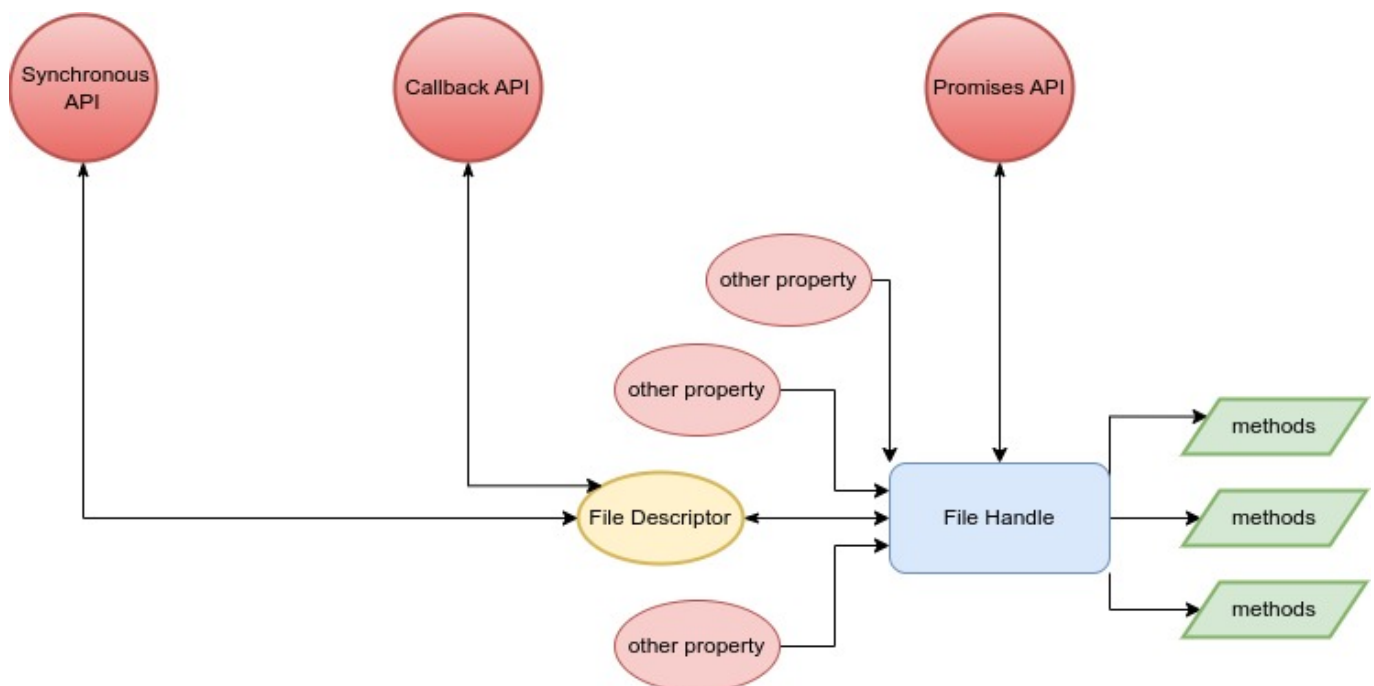
- Nuestra aplicación de NodeJs es tomada como un proceso para el sistema
- Cada proceso es administrado por el sistema operativo, y este determina la distribución y acceso que este tendrá a los recursos del sistema.
- Cada vez que un proceso interactúa con algún recurso, ya sea: hardware, software, periféricos u archivos(en este caso), el sistema reconocerá este consumo y limitará a nuestro proceso, para que no trabaje a los demás.
- Es importante administrar nosotros mismos nuestra aplicación para que no abuse del consumo de los recursos que el sistema provee.

File Descriptors: Los descriptores de archivos son un **identificador único numérico** que el sistema operativo le determina a los recursos del tipo que están siendo utilizados por un proceso. En otras palabras, son índices que le da el SO al proceso, para que este último pueda realizar sus tareas internas.

FileHandle: Es un objeto de NodeJs que se utiliza como representación del archivo que estamos operando y este guarda el **File Descriptor**. Al ser tratado como objeto este nos permite hacer uso de métodos. Además también permite, abrir un archivo, "guardarlo" en un objeto **FileHandle** y para luego pasarlo a otra función como parámetro o argumento.

El objeto FileHandle **Solo se crea y se utiliza** cuando usamos la **API de promesas** de Fs Módulo

Por lo que, tenemos que tener en cuenta cuando estamos trabajando con FileDescriptor y un FileHandle, ya que, si bien están relacionados, tienen muchas diferencias.



Buffers y Streams

Cuando hablamos de archivos y servidores, en nodejs, aparecen terminos como "buffer", "chunks" y "streams". En este curso no vamos a entrar en muchos detalles, porque son temas que se relacionan directamente con la electronica, la comunicacion y la logica computacional. Pero vamos a explicar lo necesario para entender como se aplican en el entorno de Node.

Buffer

Un buffer es un **espacio fijo** en la memoria que almacena **datos binarios**. Es similar a un array o matriz. Con datos binarios nos referimos a que guarda los datos en "crudo" o mas básicos que puede entender la PC. Estos se utilizan para **representar** información como, texto o imágenes. Si hablamos de Node, podemos manejar esta estructura de datos con la clase **Buffer** que proporciona el lenguaje JS.

Documentacion de la Clase Buffer:

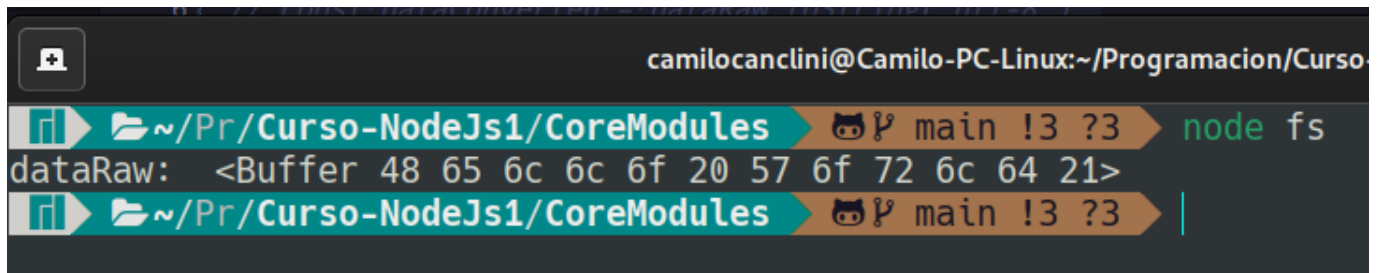
[Documentacion Oficial](#)

Si bien, el buffer, almacena datos binarios, estos se encuentran representados en hexadecimal. Esto para mejorar el rendimiento y visualización (Binario <==> Hexadecimal <==> Decimal)

En el caso de la módulo **fs**, al momento de leer los archivos, los datos llegan como un buffer en formato hexadecimal, esto podríamos tomarlo como datos en formato **RAW**.

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

const dataRaw = fs.readFileSync('data/datos.txt');
console.log('dataRaw: ', dataRaw);
```



Siguiendo, una vez que llegan, tenemos que decodificar esta información a un sistema que nosotros entendemos (El Usuario / Programador), Aqui aparecen los formatos de codificación, como **UTF-8**.

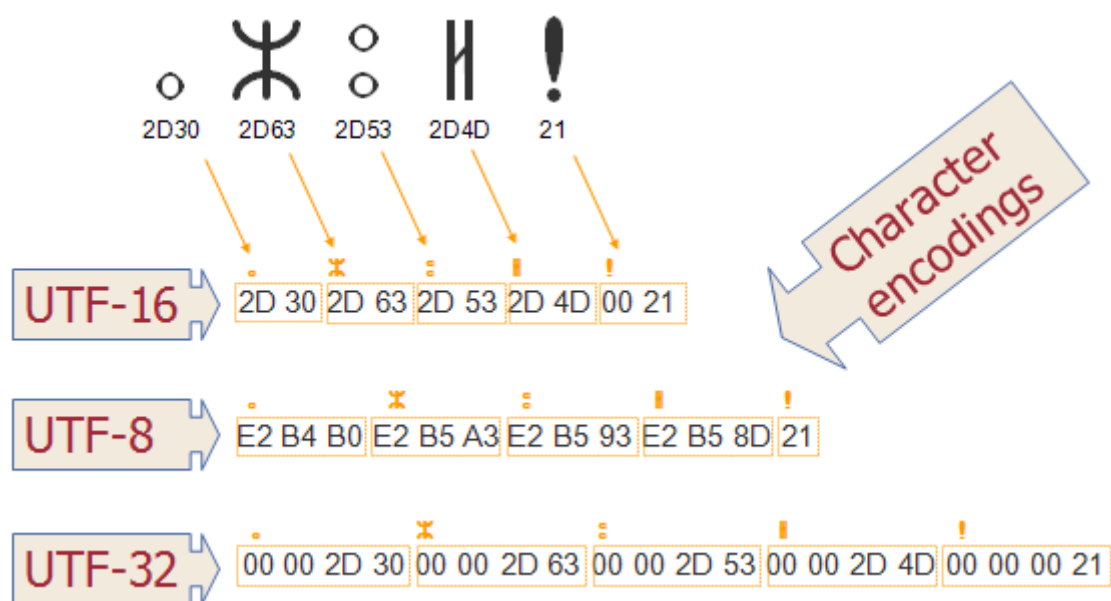
```
const fs = require('fs');

const dataRaw = fs.readFileSync('data/datos.txt');
console.log('dataRaw: ', dataRaw);

const dataConverted = dataRaw.toString('utf-8');
console.log(dataConverted);
```

```
camilocanclini@Camilo-PC-Linux:~/Programacion/Curso
~/Pr/Curso-NodeJs1/CoreModules main !3 ?3 node fs
dataRaw: <Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64 21>
dataConverted: Hello World!
```

Los formatos de codificación son los que convierten los datos del binario a las letras de nuestros respectivos lenguajes, dependiendo del lenguaje que hablemos deberemos traducirlos para un formato u otro. En Occidente el estandar es UTF-8. Siempre se recomienda establecer un formato en común para facilitar el trabajo.



Streams Y Chunks

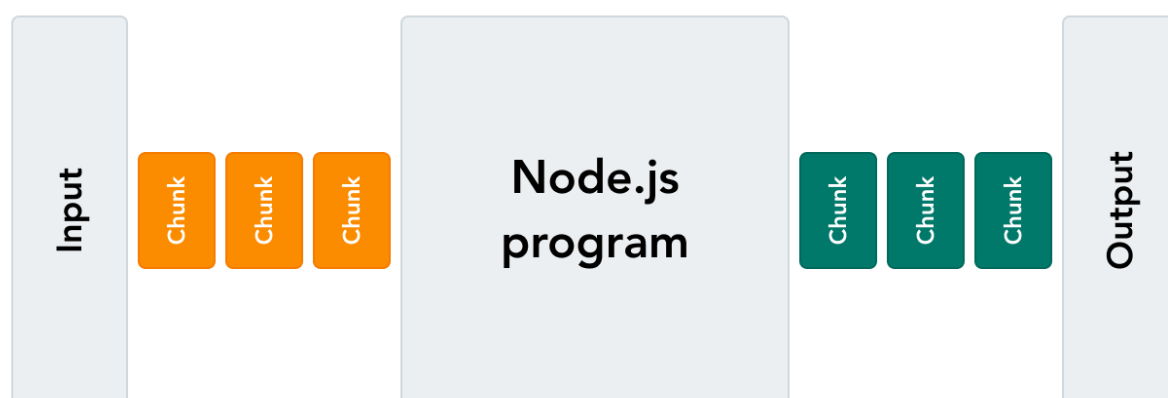
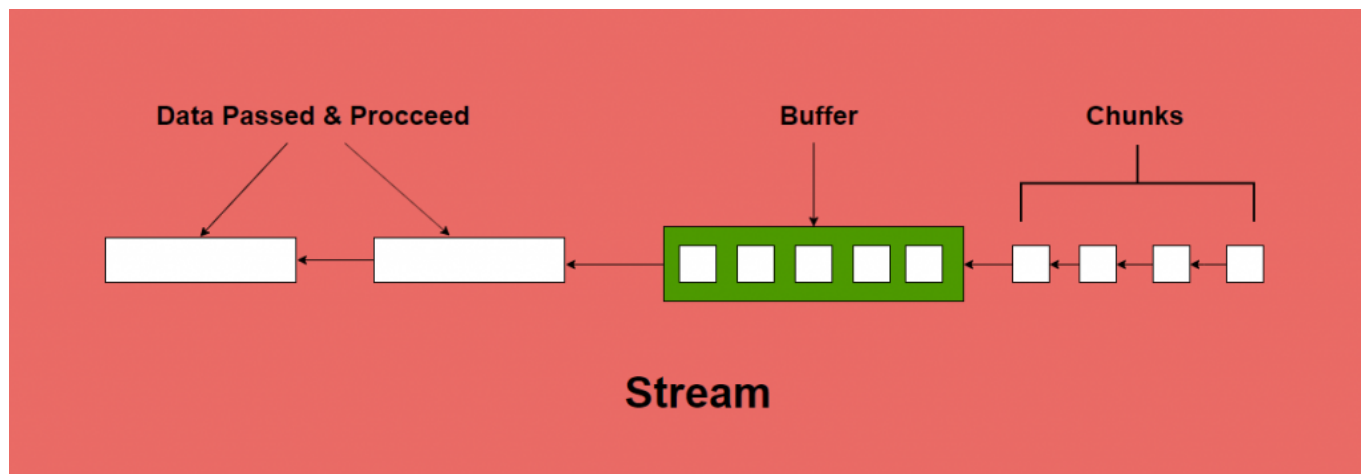
Al momento de leer y escribir tenemos que ser conscientes del tamaño de los datos que estamos manejando, ya qué, si los datos son demasiado grandes, la aplicación del servidor tenderá a ir mas lento. La forma de solucionar esto es haciendo uso de 'Streams'.

Los 'Streams' son flujos de datos, pueden ser de entrada, de salida o ambos, estos permiten recibir y enviar información en fragmentos de datos, o comunmente llamados, 'Chunks'.

Los Streams tambien tienen una clase que los representa dentro de NodeJS y nos permite operar con ellos

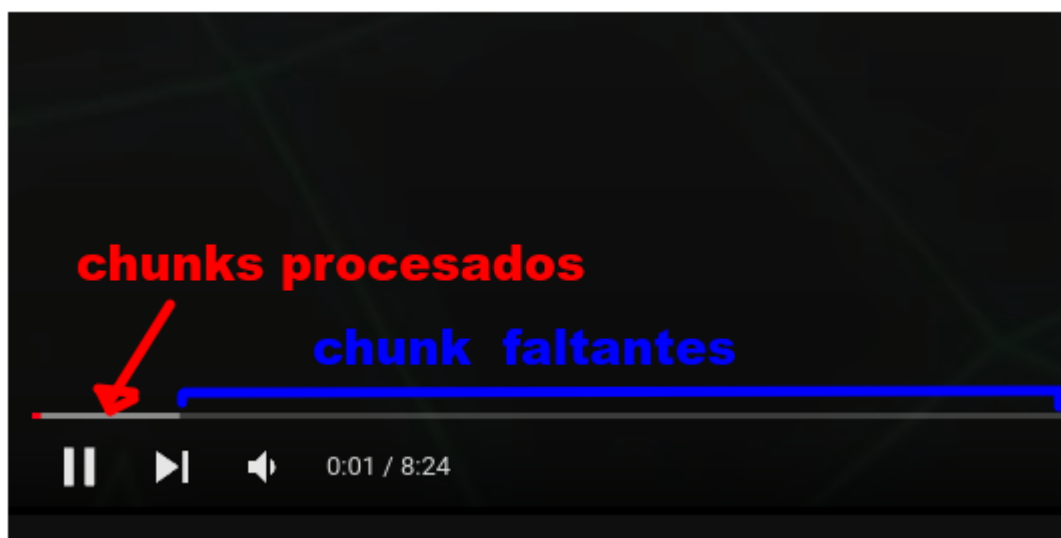
Documentacion de la Clase Stream:

[Documentacion Oficial](#)



Debido a cuestiones de rendimiento, no sería eficiente enviar toda la información de un archivo grande ni tampoco procesarla toda de un tirón. Por eso nos conviene fragmentar el archivo total en 'Chunks' e ir procesando de a poco. Un ejemplo super claro es el de YT.

Al momento de reproducir videos, YT nos envia el video de a poco para que ya podamos ir visualizandolo



El objeto Stream en Nodejs

Como dijimos anteriormente, un stream es un flujo de dato o un canal, pueden ser de entrada, de salida o ambos. Estos permiten recibir y enviar información en fragmentos de datos, o comunmente llamados, 'Chunks'.

Tenemos que aprender sobre este objeto, ya que hay mucahs librerías que trabajan con instancias del objeto stream. A su vez este objeto es una instancia de `EventEmitter`, que tambien es otra clase que se utiliza muchisimo en el entorno de Node.

Cabe acalarar que los streams tienen una API para trabajar con promesas. Esto se vera mejor en el módulo de FS

Importación

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

Tipos de Streams

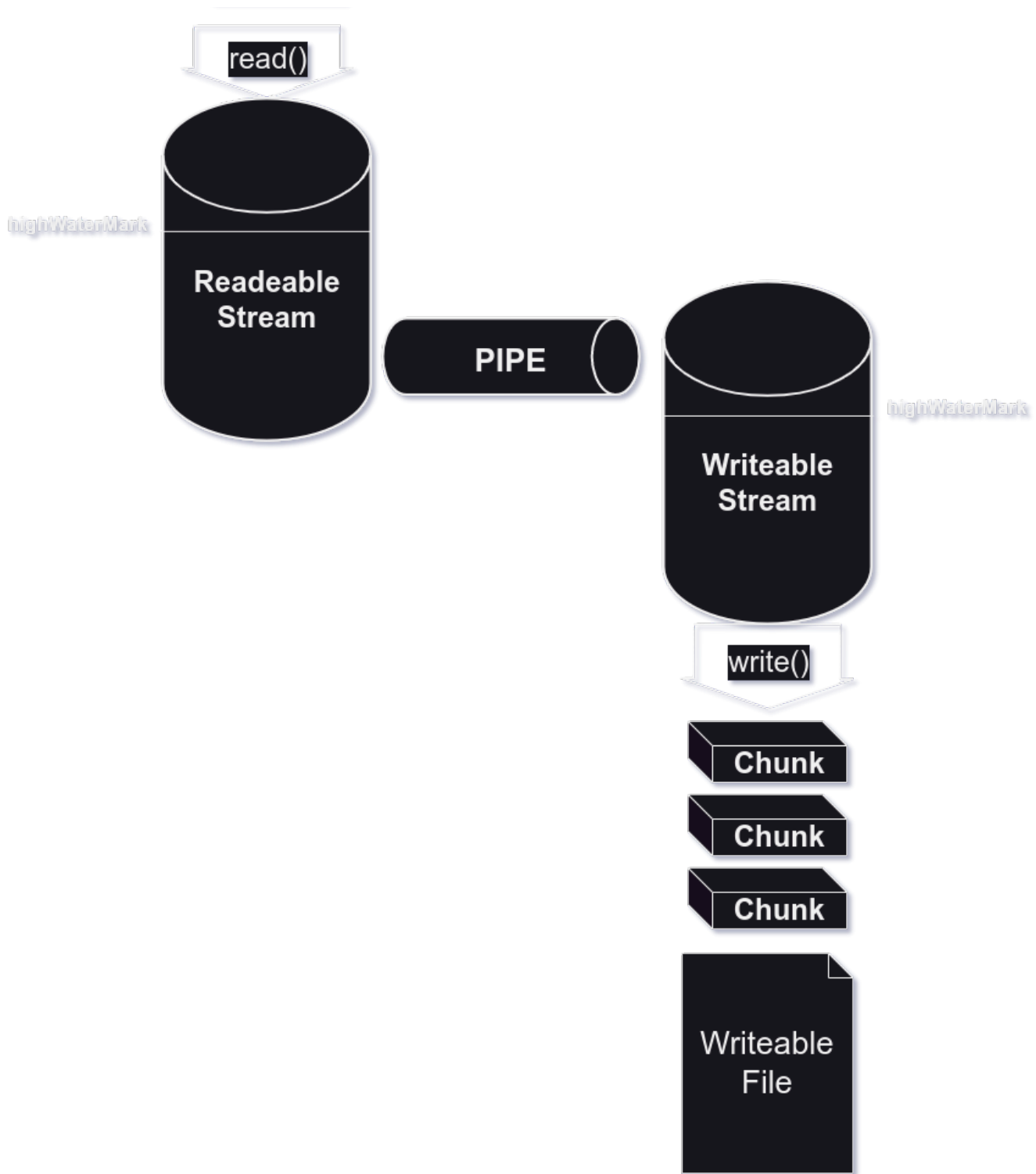
- Writable: Streams que pueden escribir la `data`.
- Readable: Streams que pueden Leer la `data`.
- Duplex: Streams que pueden escribir y leer la `data`.
- Transform: Streams que pueden escribir y leer la `data`, ademas pueden modificarla a medida que la lee o escribe.

Concepto de Buffering en Streams

Imaginemos que los streams son tanques de agua que se llenan con información y además como sabemos los `buffers` son un tipo de estructura de datos con **tamaño fijo**, por ende, este tanque de agua requiere un tope o límite. Este límite se conoce como `highWaterMark`(Marca de tope de agua).

Ahora bien, en un proceso necesitamos que los streams se comuniquen entre si para que el sistema entero funcione, para comunicarlos utilizaremos los `pipelines`. Estos no son mas que un metodo propio de la libreria `stream` que permite conectar dos streams.





Si los visualizamos como tanques de agua, el concepto se ve entiende a simple vista. Al momento de que se alcanzan los **highWaterMarks** los streams se regulan por si solos, dependiendo del modo en el que se encuentren trabajando, esto para no abusar de la memoria y para sincronizarse con su par para mantener un flujo de datos aceptable.

A continuación vamos a ver cada tipo de stream con sus metodos y eventos correspondientes, he aqui un pequeño resumen de los mismos y seguido, un lista de las los objetos que los utilizan.

Readable Streams

Events

- data
- end
- error
- close
- readable

Functions

- pipe(), unpipe()
- read(), unshift(), resume()
- pause(), isPaused()
- setEncoding()

Writable Streams

Events

- drain
- finish
- error
- close
- pipe/unpipe

Functions

- write()
- end()
- cork(), uncork()
- setDefaultEncoding()



Streams

Readable Streams

- HTTP responses, on the client
- HTTP requests, on the server
- fs read streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdout and stderr
- process.stdin

Writable Streams

- HTTP requests, on the client
- HTTP responses, on the server
- fs write streams
- zlib streams
- crypto streams
- TCP sockets
- child process stdin
- process.stdout, process.stderr



Antes de explicar cada uno de los streams, recuerda que la clase `Stream` esta directamente relacionada con la clase `EventEmitter`. (Comparten métodos)

Writable Streams

Instanciación manual

Eventos Writable Streams

- **close**: Este evento cuando el archivo que estamos escribiendo se cierra, esto lo veremos mejor en el módulo fs.
- **drain**: Este evento se dispara cuando podemos continuar la escritura con **write()**, recordemos el concepto del buffering.
- **error**: Como su mismo nombre indica, y vimos, se dispara cuando ocurre un error de escritura o al "conectar" un pipe.
- **pipe/unpipe**: Estos 2 eventos son opuestos, el primero se dispara cuando por ejemplo un **Stream Readable** se "conecta" con un pipe a nuestro **Writable Stream**, y el segundo se dispara cuando el pipe se "desconecta".
- **finish**: Este evento se dispara cuando se escribe el ultimo pedazo de información, concretamente despues de que se utiliza el método **writable.end()**

Métodos Writable Streams

Writable = Stream.Writable instanciado *Recorda que cuando un método o función tiene sus argumentos encerrados entre corchetes quiere decir que estos son opcionales*

- **writable.write(chunk[, encoding][, callback])**: Este método permite escribir los datos entrantes **en el Stream**. Tiene un callback que permite ingresar el argumento **err** para manejar errores.
- **writable.end([chunk[, encoding]][, callback])**: Este método permite escribir un ultimo dato en el stream , **despues de ser llamado NO se puede llamar de vuelta a writable.write()**
- **writable.cork()**: Este método lo que hace es almacenar todos los datos **siguientes** que se escriban con **.write()** en memoria. Esto se usa para almacenar los chunks pequeños entrantes, ya que estos pueden relentizar en flujo de datos y por ende, tal vez, es mejor procesarlos despues.
- **writable.uncork()**: Es el método que saca lo datos almacenados previamente en memoria por **.cork()**. Si **cork()** fue llamado multiples veces, entonces **uncork()** debe ser llamado el mismo numero de veces.

Readable Streams

Para empezar este tipo a streams tienen 2 modos:

- El **Flowing Mode**: Los datos se leen automatica e inmediatamente, y además se utilizan events emitters
- Y el **Paused Mode**: Para leer los datos se necesita usar **stream.read()** manualmente.

Todos los streams de este tipo comienza en **Pause Mode**. Para convertirlos a **Flowing Mode** tenemos 3 formas:

1. Usando el Event **data**, lo vemos a continuación.
2. Ustando el método **stream.resume()**, lo vemos a continuación.

3. Usando `stream.pipe(writableStream)`, lo vemos a continuación.

y para pasar a **Paused Mode**:

1. Usando el método `stream.pause()`
2. Desconectando todos los pipes del stream, con el método `stream.unpipe()`

Ademas cada `readableStream` posee 3 estados internos, los cuales controlan indican el flujo de datos, esta propiedad se llama `readableFlowing`.

- `readable.readableFlowing === null`: Este estado significa que no hay fuentes para leer los datos o que no hay destino a donde enviarlos.
- `readable.readableFlowing === true`: Este estado significa que sí hay una fuente (event listener para `data`), destino (`readable.pipe()`) o si se llamó a `readable.resume()` y se encuentra a la espera de datos.
- `readable.readableFlowing === false`: Este estado signifca que hay fuente y/o destino para los datos pero llamó a `readable.pause()`, `readable.unpipe()`

Eventos Readable Stream

- **data**: Este evento se dispara cuando se detecta que hay información esperando por ser leída (Chunks)
- **end**: Este evento se dispara cuando **ya no hay mas datos** que consumir
- **close**: Este evento cuando el archivo que estamos leyendo se cierra, esto lo veremos mejor en el módulo fs.
- **error**: Como su mismo nombre indica, y vimos, se dispara cuando ocurre un error de escritura o al "conectar" un pipe.
- **readable**: Este evento se dispara cuando hay datos que leer o cuando se llega al final. Si ya no hay nada que leer `stream.read()` devuelve null

Métodos Readable Stream

- `readable.read([size])`: Este método lee los datos del buffer interno y los retorna, si no hay ninguna codificación seteada entonces devuelve un buffer. Podemos ver tambien que su argumento opcional es `size`, este representa la cantidad de bytes maximos que leera, pero de vuelta, es opcional.
- `readable.pipe(destination[, options])`: Permite conectar un `writableStream`, este cambiará al **Flowing Mode** y administrará el envio de los datos almacenados a el nuevo destino. Por otro lado, tambien retornará el `writableStream`, esto para poder concatenar otros metodos seguido de este. *Es posible conectar un readableStream a varios writableStreams.*

Cuando ocurre un error en el `readableStream`, el `writableStream` seguira abierto, por lo que, es necesario cerrarlo manualmente.

- `readable.unpipe([destination])`: Para desconectar los pipes que establecimos con el anteriormente usamos este método. Además, como podemos ver, admite un solo argumento que es opcional, aquí va el `writableStream` que queremos desconectar, si no especificamos nada se van a desconectar todos los pipes conectados.
- `readable.unshift(chunk[, encoding])`: Este método permite ingresar datos al principio del buffer interno de un `readableStream`.
- `readable.pause()`: Este método hace que el stream deje de emitir el evento `data` y cambia al Pause Mode. Algo a acalarar es que, si se escucha el evento `readable`, este método no hará efecto.
- `readable.resume()`: Este método hace lo opuesto al anterior. Tampoco tendrá efecto si se dispara el evento `readable`.

Los Eventos (Event Handlers - Event Emitters - Events Listeners)

Documentación Oficial Events

Aquí se encuentra toda la información del módulo:

Documentacion Oficial

Paradigma de Eventos

A la hora de programar en JS tenemos que ser conscientes de que gran parte de los metodos y objetos que manejamos en este lenguaje tienen propiedades o se relacionan con los eventos.

Este paradigma de la programación hace referencia a un modelo ASÍNCRONO en el cual existen objetos o entidades que DISPARAN EVENTOS en la aplicación y otros los cuales quedan a la ESCUCHA de estos, estos 2 grupos se conocen como: `Event Emitters` y `Event Listeners` respectivamente.

En Node tenemos que conocer aunque sea el concepto y los métodos básicos que se utilizan en este modelo, ya que como dijimos, muchos módulos y objetos que se utilizan en Node hacen uso de eventos.

Clase EventEmitter

Para crear un disparador de eventos, podemos hacerlo usando instanciando un objeto con la clase `EventEmitter`

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

EventEmitter

Y para emitir un evento usamos `myEmitter.emit(nombreDelEvento[, , args])`. Como podemos ver, el método permite pasar tantos argumentos como veamos necesarios.

```
myEmitter.emit('elEvento', 'arg1', 'arg2', 'argFina');
```

EventListeners

Ahora, para escuchar los eventos emitidos podemos hacer uso de los métodos:

- `myEmitter.on(event, callback)`: Ejecuta una función callback **cada vez** que se detecta un evento, el evento que especifiquemos tiene que ser con un string.
- `myEmitter.once(event, callback)`: Ejecuta una función callback la **primera vez** que se dispara el evento.
- `myEmitter.addListener`: Hace exactamente lo mismo que `myEmitter.on()`, son sinonimos.

Ejemplo de Código

```
//Importación y declaracion de Emitter
const {EventEmitter} = require('events');
const myEmitter = new EventEmitter();

myEmitter.on('saludar', (arg1, arg2) => {
  console.log('Hola!', ' ', arg1, ' ', arg2);
});
myEmitter.on('despedir', (arg1, arg2) => {
  console.log('Chau!', ' ', arg1, ' ', arg2);
})

// Emisión de Eventos y Pasando Argumentos para las funcioens
myEmitter.emit('saludar', 'Camilo', 'Canclini');
myEmitter.emit('despedir', 'Camilo', 'Canclini');

console.log(myEmitter.eventNames()); // Muestra los eventos escuchados en el
emitter
console.log(myEmitter.listeners('saludar')); // Muestra las funciones
Listeners asociadas a este evento

// Despues de un tiempo le saca la escucha de un evento y vuelve a ejecutar
las emisiones
setTimeout(() => {

  //Remueve todas las funciones Listener Asociadas al evento saludar
  myEmitter.removeAllListeners('saludar');

  myEmitter.emit('saludar', 'Camilo', 'Canclini');
  myEmitter.emit('despedir', 'Camilo', 'Canclini');

}, 1000)
```

Diferencias de eventos entre NodeJS y JS ejecutado en la web(JSDOM)

Tenemos que saber diferenciar las formas de operar de cada uno, porque los eventos, como concepto, tienen propiedades diferentes dependiendo el contexto.

Por ejemplo, cuando nos referimos al DOM del navegador, existe una jerarquía que relaciona a todos los elementos de la pagina que estamos desarrollando.

Al momento de dispararse un evento, este se propaga entre los diferentes elementos relacionados con el `target`. El `target` no es ni mas ni menos que el objeto que dispara esta notificación.

Cada objeto se basa en la interfaz de `eventTarget`, con la cual cada uno define sus propios eventos, metodos y propiedades.

Ahora cuando hablamos de NodeJS, esta propagación no existe, porque no existe una jerarquía, entre objetos. Ademas los objetos que definimos que disparan eventos son instancias de la clase `EventEmitter`.

En conclusión, son modelos diferentes, que se utilizan para contextos diferentes, ya que, no es lo mismo estar trabajando con los objetos del DOM que con objetos propios creados desde el entorno de NodeJS.

Igualmente cabe recalcar que Node ofrece en su módulo de `events` una clase que permite emular, por así decirlo, a los `eventTarget` del DOM, la clase se llama: `nodeEventTargets`.

Core Modules NodeJS

Ahora vamos a ver los modulos principales que vienen integrados con Node, estos son las bases de los frameworks y demas librerías que se utilizan hoy en día. Estos nos van a permitir, acceder a los recursos del sistema directamente desde JS. **De cada módulo se mostraran los métodos mas relevantes y ademas habrá en la carpeta 'Core Modules' un ejemplo de uso de cada uno**

OS Module

Este modulo nos permite obtener datos del hardware y software del equipo que esta ejecutando nuestro script:

Documentación Oficial OS

Aquí se encuentra toda la información del módulo: [Documentacion Oficial](#)

Importación OS

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

Métodos OS

```
os.totalmem() //Devuelve INT de la memoria total
os.freemem() // Devuelve INT de la memoria libre
os.getPriority() // Devuelve un INT con el valor de PRIORIDAD del proceso actual
os.homedir() //Devuelve un STRING de la ruta del directorio del usuario
os.tmpdir() //Devuelve un STRING de la ruta del directorio temporal
```

```
os.hostname() //Devuelve un STRING con el nombre del equipo
os.platform() //Devuelve un STRING con el nombre del sistema operativo
os.uptime() //Devuelve un ENTERO del tiempo que a estado corriendo la
maquina desde que se encendió
os.networkInterfaces() //Devuelve un OBJETO con la informaci3n del
adaptador de red
os.userInfo() //Devuelve un OBJETO con la informaci3n del usuario actual
```

PATH Module

Este m3dulo permite trabajar con las rutas de directorios y archivos del sistema operativo en el que nos encontremos. Recordemos que NodeJS, es multiplataforma, puede estarce ejecutando tanto en un windows como en un linux, por lo que, las formas y privilegios para acceder a los disntintos recursos del SO cambian, entre uno y otro.

Documentaci3n Oficial PATH

Aqu3 se encuentra toda la informaci3n del m3dulo: [Documentacion Oficial](#)

Importaci3n PATH

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

M3todos PATH

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

//Devuelve un STRING con el separador que utiliza el SO para los PATHS
// WINDOWS --> \
// LINUX --> /
path.sep;

//Retorna un STRING de la 3ltima parte de un path
path.basename('/home/camilocanclini/Documents/hola.txt');
//Retorna un STRING pero recortando lo que indiquemos en el segundo
argumento
path.basename('/home/camilocanclini/Documents/hola.txt', '.txt');
//Devuelve un STRING del path sin la ultima parte
path.dirname('/home/camilocanclini/Documents/hola.txt');

//Devuelve un STRING con el separador que usa el OS para separar los paths
// WINDOWS --> ;
// LINUX --> :
path.delimiter;

//Devuelve un STRING que indica la extensi3n del path
```

```

path.extname('hola.txt');

//Devuelve un STRING, este genera un PATH haciendo uso de los argumentos
que pasemos
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..');

//Devuelve un STRING, este genera un PATH NORMALIZADO para el SO en que nos
enconetremos
//Que este normalizado quiere decir que es valido, que puede ser reconocido
por nuestro SO
path.normalize('/foo/bar//baz///asdf/quux/..');

//Devuelve un OBJETO con las propiedades del PATH que pasemos como
argumento
path.parse('/home/user/dir/file.txt');
const pathObj = path.parse('/home/user/dir/file.txt');

//Hace la operacion opuesta que path.parse
path.format(pathObj);

/*
Hace lo mismo que path.join pero ademas tienen en cuenta
si los argumentos pasados son PATHS ABSOLUTOS o RELATIVOS

PATH ABSOLUTO --> /hola/como/estas
PATH RELATIVO --> hola/como/estas ó ./hola/como/estas

Este metodo analiza de derecha a izquierda
Si es absoluto se toma ese path y desde ahi, hacia la derecha se unen los
demas paths
Si NO HAY PATHS ABSOLUTOS se agrega el PATH total y se unifica con los
pasados por Argumentos
*/
path.resolve('hola/pepe', 'argentina/moni');
path.resolve('/hola/pepe', 'argentina/moni');
path.resolve('hola/pepe', '/argentina/moni');

```

Constantes del modulo: `__filename` y `__dirname`

Las constantes del modulo son parecidas a los `global objects`, ya que cada modulo o script puede llamarlas, pero el valor dependera del script que la llame.

En concreto, las que presentaremos a continuación, son para trabajar con los PATHs del script en cuestion.

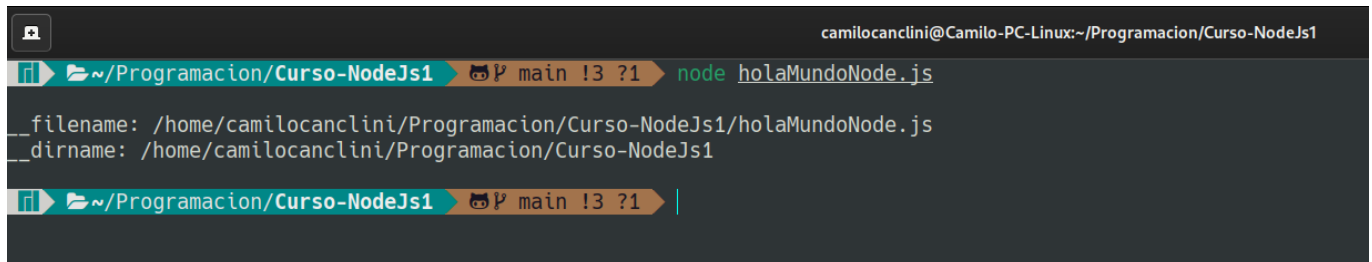
- `__filename`: Esta devuelve el PATH ABSOLUTO o RUTA ABSOLUTA del archivo.
- `__dirname`: Esta devuelve el PATH ABSOLUTO o RUTA ABSOLUTA del directorio que contiene al archivo

Ejemplos:

Codigo:

```
console.log('\n__filename:', __filename);
console.log('__dirname:', __dirname, '\n');
```

Resultado:



```
camilocanclini@Camilo-PC-Linux: ~/Programacion/Curso-NodeJs1
node holaMundoNode.js
__filename: /home/camilocanclini/Programacion/Curso-NodeJs1/holaMundoNode.js
__dirname: /home/camilocanclini/Programacion/Curso-NodeJs1
```

Estas constantes se suelen utilizar a la hora de confeccionar PATHS o para el manejo de archivos.

FS Module

Este módulo nos permite interactuar con los archivos del sistema, nos da las herramientas para modificarlos. Para comenzar tenemos que saber que este módulo nos permite interactuar con los archivos de 3 formas diferentes, dentro del entorno se las conoce como API (Aplication Program Interface | Interfaz de Programacion de la Aplicación), estas no son como las APIs WEB que comunmente se utilizan hoy en día. Simplemente pensemos que son las **formas en las cuales podemos interactuar con los archivos**

A continuación vamos a presentar 3 ejemplos en los cuales se importa la misma función con la forma de EMAScript6(`import { unlinkSync } from 'node:fs';`). En el primero vamos a trabajar de forma sincrónica(la mas simple, pero tambien la mas vulnerable), en la segunda, vamos a usar callbacks y en la tercera vamos a usar promesas. De estas 2 últimas ya hemos hablado en profundidad anteriormente.

API para Sincronía

```
import { unlinkSync } from 'node:fs';

try {
  unlinkSync('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (err) {
  // handle the error
}
```

API para Callbacks (Asincronía)

```
import { unlink } from 'node:fs';

unlink('/tmp/hello', (err) => {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

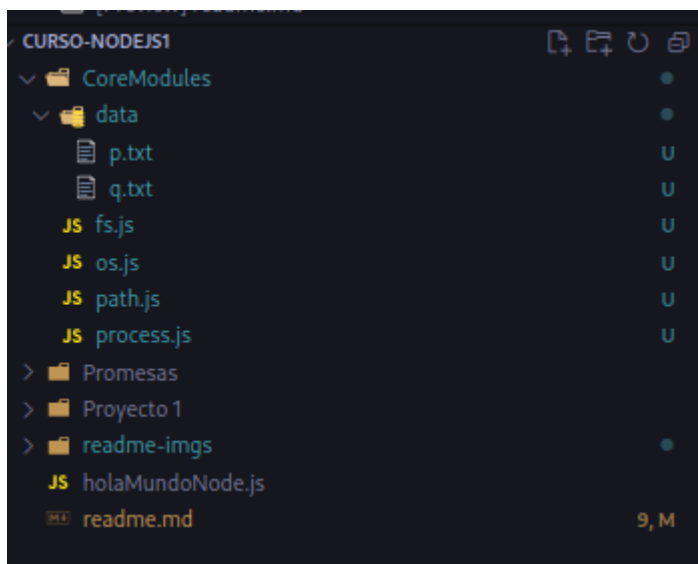
API para Promesas (Asincronía)

```
import { unlink } from 'node:fs/promises';

try {
  await unlink('/tmp/hello');
  console.log('successfully deleted /tmp/hello');
} catch (error) {
  console.error('there was an error:', error.message);
}
```

Como podemos apreciar este módulo nos permite elegir si el método se ejecutará de manera asíncrona o síncrona. Como ya vimos, si trabaja de forma síncrona las líneas de código siguiente no se ejecutarán hasta que hayamos terminado la operación con el método indicado.

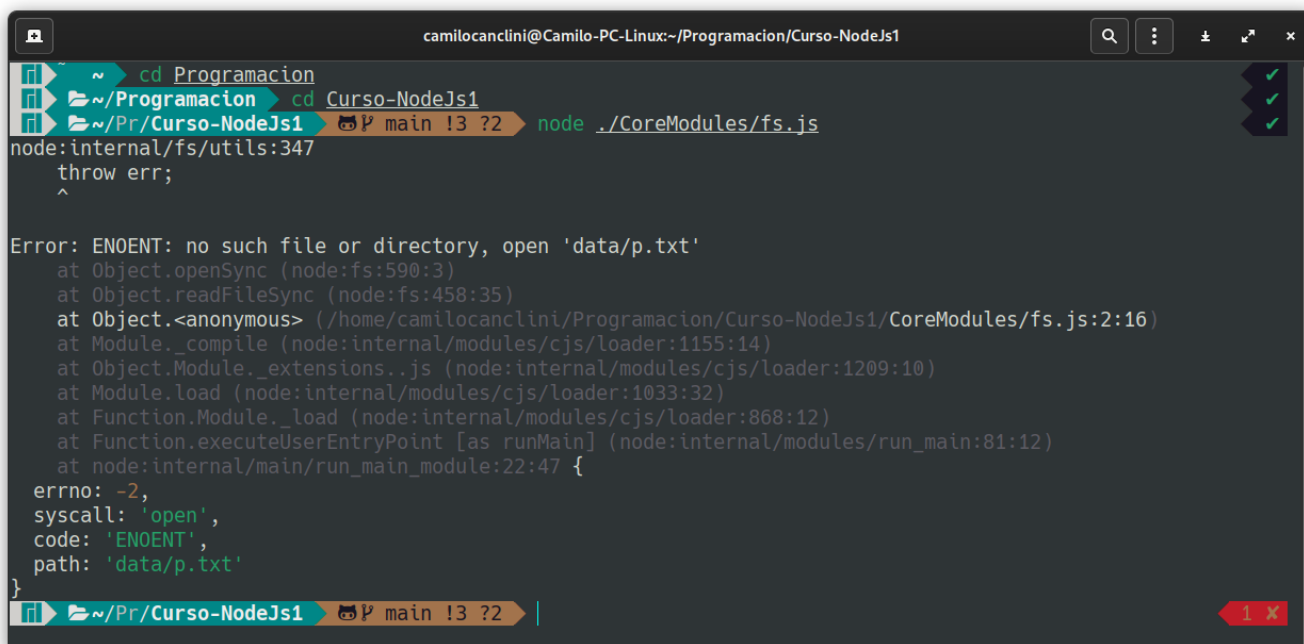
Por último algo a tener en cuenta es que cuando definimos la rutas de los archivos con los que interactuamos tenemos que saber que las rutas son relativas con respecto al **Working Directory** y no con respecto al archivo desde el cual estamos ejecutando. Por ejemplo, vease la siguiente estructura:



y ahora vea el siguiente código del archivo llamado `fs.js`

```
const fs = require('fs');
console.log(fs.readFileSync('data/p.txt'));
```

Si yo ejecuto el script `fs.js` de la siguiente forma, voy a recibir un error:



```
camilocanclini@Camilo-PC-Linux:~/Programacion/Curso-NodeJs1
~ cd Programacion
~/Programacion cd Curso-NodeJs1
~/Pr/Curso-NodeJs1 main !3 ?2 node ./CoreModules/fs.js
node:internal/fs/utils:347
  throw err;
  ^

Error: ENOENT: no such file or directory, open 'data/p.txt'
    at Object.openSync (node:fs:590:3)
    at Object.readFileSync (node:fs:458:35)
    at Object.<anonymous> (/home/camilocanclini/Programacion/Curso-NodeJs1/CoreModules/fs.js:2:16)
    at Module._compile (node:internal/modules/cjs/loader:1155:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1209:10)
    at Module.load (node:internal/modules/cjs/loader:1033:32)
    at Function.Module._load (node:internal/modules/cjs/loader:868:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:22:47 {
  errno: -2,
  syscall: 'open',
  code: 'ENOENT',
  path: 'data/p.txt'
}
```

¿Por qué ocurre esto?, porque al momento de yo ejecutar con `node ./CoreModules/fs.js` mi "Working Directory" es `.../Curso-NodeJS1/` por lo que cuando el script `fs.js` ejecuta la linea 2 (`console.log(fs.readFileSync('data/p.txt'));`) busca de la siguiente manera --> `/Curso-NodeJS1/data/p.txt` y esa ruta no existe, porque no pasa por la carpeta `CoreModules`. ¿Como se soluciona?, tenemos 2 formas:

1. O bien cambiamos la ruta del script quedandonos:

```
console.log(fs.readFileSync('CoreModules/data/p.txt'));
```

2. O cambiamos el **Working Directory**: haciendo `cd CoreModules/` y despues ejecutamos `node fs.js`

En el caso del segundo ya se volveria valida la ruta que especificamos en la segunda linea.

Documentación Oficial FS

Aquí se encuentra toda la información del módulo: [Documentacion Oficial](#)

Importación FS

```
const fs = require('fs');
const fsPromise = require('fs/promises');
```

Abrir y Cerrar Archivos

Siguiendo con temas de optimización, al momento de abrir y cerrar archivos tenemos que ser conscientes que, la misma acción de abrir y cerrar, consume recursos. Por lo que cuando veamos a continuación los métodos, vamos a dividirdelos en 2 grupos:

- Los métodos del primer grupo son todos aquellos que abren y cierran el archivo al terminar de ejecutarse

- Y los metodos del segundo grupo, corresponden a los que necesitan que el archivo se encuentre abierto manualmente con el método `fs.open()`.

Los del segundo grupo serán mas eficientes, ya que nos permiten ejecutar varias operaciones en un archivo sin al necesidad de estar abriendo y cerrando cada vez que ejecutamos una operación.

En otras palabras, **solo se abre y se cierra una vez**. La unica contra es que, no debemos olvidar cerra el archivo con `fs.close()`

Métodos FS

Recordemos que las 3 maneras de programar con FS (Synchronous, Callbacks y Promises) comparten la mayoría de los metodos y lo unico que cambia es la forma de llamar a ese método.

Para este caso vamos a presentar todos los métodos con la forma de callback:

Grupo de Metodos 1 (Abren y Cierran el archivo cada vez que se ejecutan)

```
const fs = require('fs');
const { exit } = require('process');

// +-- " G R U P O 1 " --+

// Leer
fs.readFile('./data/elArchivo.txt', (err,data) => {
  if (err) console.log('Ocurrió un error al leer');
  console.log('1:',data);
  console.log('2:',data.toString());
  // Leer Archivos y convertirlos antes (UTF-8)
});

// Crear y Editar

// Se modifica el archivo entero
fs.writeFile('./data/nuevoArchivo.txt','Hola! Soy un archivo nuevo',
(err)=>{
  if (err) throw Error('Hola che');
  console.log('3: el archivo fue creado si no existia, y se editó');
});

//Se agrega información nueva
fs.appendFile('./data/nuevoArchivo.txt', '\nHola!', (err)=>{
  if (err) console.log('Hola che');
  console.log('4: el archivo fue creado si no existia, y se agregó info nueva');
});

// Chequea si el archivo es accesible
// (Si existe, si puede ser leído, escrito o ejecutado
// Se usan constants:
// F_OK: Verifica si el archivo existe
```

```
// R_OK: Verifica si el archivo puede leerse
// W_OK: Verifica si el archivo puede escribirse
// X_OK: Verifica si el archivo puede ejecutarse
fs.access('./data/datos.txt',fs.constants.F_OK, (err) =>{
if (err) {
    console.log('Ocurrió un error al verificar el acceso al archivo')
}else{
    console.log(' 5: El archivo existe (En este caso [F_OK])');
}
})

// Permite copiar archivos
// Este método también acepta constantes para cambiar el comportamiento del
método
// (Ver las constantes en la documentación oficial)
fs.copyFile('./data/elArchivo.txt','./data/elArchivoCopiado.txt',
fs.constants.COPYFILE_EXCL,(err) => {
    if (err) {
        console.log('Ocurrió un error al copiar el archivo');
    } else {
        console.log('6: El archivo se copió');
    }
})

//Devuelve información (Metadatos) del archivo
fs.stat('./data/datos.txt',(err, stats)=>{
    if (err) {
        console.log('Ocurrió un error al obtener metadatos');
    } else {
        console.log('7: metadatos: ', stats);
        console.log('7: Tamaño del Archivo: ', stats.size, 'bytes');
    }
})

//Crea un directorio
fs.mkdir('./data/nuevoDirectorio',(err)=>{
    if (err) {
        console.log('Ocurrió un error al crear directorio');
    } else {
        console.log('8: El directorio se creó correctamente');
    }
})

// Abre un directorio
fs.opendir('./data',(err,dir)=>{
    if (err) {
        console.log('Ocurrió un error al abrir el directorio');
    } else {
        console.log('9: dir: ', dir);
    }
})

// Borra un archivo o un directorio
fs.rm('./data/nuevoArchivo.txt',(err)=>{
```



```

    if (err) {
        console.log('Ocurrió un error al borrar el directorio');
    } else {
        console.log('10: Se borró el directorio (nuevoArchivo)');
    }
})

// Trunca un archivo (Vacía el archivo, borra todo el contenido)
fs.truncate('./data/nuevoDirectorio/archivoTruncado.txt', (err)=>{
    if (err) {
        console.log('Ocurrió un error al truncar el archivo');
    } else {
        console.log('11: Archivo Truncado exitosamente');
    }
})

setTimeout(()=>{
    console.log('Empezando a observar el archivo');
    console.log('Realice algun cambio en el archivo (Tiene 10 seg)');

    // Dispara un evento cuando el archivo es alterado
    // Es un loop, se queda escuchando si ocurren cambios en el archivo y
    ejecuta el callback
    fs.watch('./data/nuevoDirectorio/archivoObservado.txt', (event, file)
=>{
        console.log('12: El archivo: ', file, 'Fué modificado');
        console.log('12: El evento que se disparó fue: ', event);

        console.log('Cerrando programa');
        process.exit();
    })

    setTimeout(()=>{
        console.log('\n\nNo se realizaron cambios en el archivo');
        console.log('Cerrando programa');
        process.exit();
    },10000);
},4000);

```

Grupo de metodos 2 (`fs.open()`, se habren y se cierran 1 sola vez)

```

// +-- " G R U P O 2 " --+

const fs = require('fs');

// Abriendo el Archivo
fs.open('./data/elArchivo.txt','r+',(err,fd)=>{
    if (err) {
        console.log('Fallo al abrir archivo: ',err.message)
    } else {

```

```

// Leyendo el archivo abierto previamente
fs.read(fd, (err,br,bufferobj)=>{
  if (err) {
    console.log('Fallo al leer archivo: ',err.message)
  } else {
    console.log(`\nSe leyeron ${br} bytes del buffer\n`);
    console.log(`El buffer traducido:\n${bufferobj}\n`);
    console.log(bufferobj.byteLength);

    // Una vez leído se escribe el archivo
    fs.write(fd, 'Escribiendo...',(err,bw,buffer)=>{
      if (err) {
        console.log('Fallo al escribir archivo:
',err.message)
      } else {
        console.log(`\nSe escribieron ${bw} bytes del
buffer\n`);
        console.log(`El buffer traducido:\n${buffer}\n`);

        // Una vez que se terminó de escribir se cierra el
archivo
        fs.close(fd, ()=>{
          console.log('El archivo se cerró');
        });
      }
    });
  }
});
}
});
}
});

```

Miremos que en el segundo grupo de métodos se esta haciendo uso del fileDescriptor (`fd`). Además miremos que al momento de abrir el archivo con `fs.open(path, options[flag, ...], callback)`, podemos pasar como argumento una `flag` o `mode` de apertura.

Esto lo que indica es el tipo de operaciones o el alcance de las mismas, que tenemos para realizar las operaciones una vez abierto el archivo, vease la siguiente tabla:

Flag	Description
<code>r</code>	To open file to read and throws exception if file doesn't exists.
<code>r+</code>	Open file to read and write. Throws exception if file doesn't exists.
<code>rs+</code>	Open file in synchronous mode to read and write.
<code>w</code>	Open file for writing. File is created if it doesn't exists.
<code>wx</code>	It is same as 'w' but fails if path exists.
<code>w+</code>	Open file to read and write. File is created if it doesn't exists.
<code>wx+</code>	It is same as 'w+' but fails if path exists.
<code>a</code>	Open file to append. File is created if it doesn't exists.
<code>ax</code>	It is same as 'a' but fails if path exists.
<code>a+</code>	Open file for reading and appending. File is created if it doesn't exists.
<code>ax+</code>	It is same as 'a+' but fails if path exists.

PROCESS MODULE

Este módulo permite obtener información y controlar el proceso actual que genera nuestra aplicación en el sistema operativo.

Que es CLI y Command Line Apps

El concepto CLI significa Command Line Interface, esta es simplemente la forma mediante la cual el usuario se comunica con el sistema operativo vía la linea de comandos. Dependiendo del Sistema Operativo estas varían, por ejemplo en Windows se utiliza powershell y en linux bash, aunque estas pueden cambiar.

Gracias a CLI es que podemos ejecutar comandos como `node app.js`, lo que ocurre aquí es que, le estamos diciendo a la consola de comandos que queremos usar el programa `node` y le pasamos un argumento que es la ruta del script que va a procesar.

Esta forma simple de trabajar e interactuar con el sistema permite la crear aplicaciones que trabajen haciendo uso de la consola. Aquí aparecen las Command Line Apps, que basicamente son, aplicaciones que reciben instrucciones o argumentos desde la consola y ejecutan alguna tarea en base a esos parametros, o pueden responder con texto via la misma interfaz de comandos.

NodeJS permite la creación de este tipo de aplicaciones. El siguiente módulo permitira tener un acercamiento con este concepto.

Este módulo permite manejar eventos que ocurren durante la ejecución del proceso, vea la sección de métodos

Enviroment Variables

Cuando hablamos de consola y en general de sistemas operativos, aparecen las variables de entorno. Para no entrar en muchos detalles diremos que son, variables que almacenan información de configuracion para la sesion en la que nos encontramos. Por ejemplo, en el sistema operativo, estas guardan:

- El nombre del usuario actual
- Las rutas a las carpetas importantes del sistema
- Datos del sistema operativo
- Crendenciales de usuario y de programas

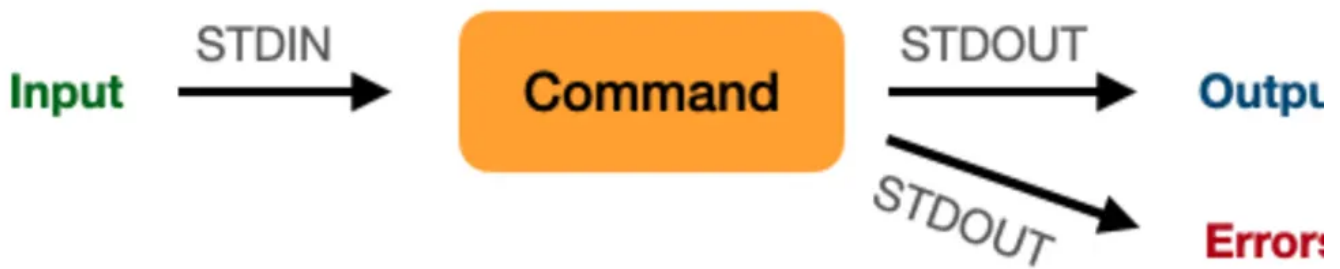
Se utilizan a traves del modulo process y otros paquetes, para almacenar y recuperar información para nuestra aplicación. Por ejemplo, podriamos obtener el nombre del usuario desde el sistema y saludarlo con un mensaje personalizado.

Para el proposito de este curso las utilizaremos para recuperar información de configuracion de nuestra aplicación, esto sera visto mas adelante.

Las entradas y Salidas (Standard)

Otro concepto que aparece cuando hablamos de consola son los flujos de datos estándar, estos no son mas que Streams (Los que vimos antes) proporcionados por el sistema operativo. Se dice que son estándar ya que todos los sistemas lo tienen. Concretamente son, interfaces que nos permiten ingresar información y sacar información de un proceso o aplicación. Si son Streams, tambien aparcent los famosos "Pipes".

Un ejemplo que ilustra bastante bien esto son los programas de consola de linux, los cuales permiten conectar procesos, ¿Cómo?, simplemente uniendo la salida de un proceso con la entrada del otro, usando un pipe



Para lo que nosotros los podemos usar es para crear Aplicaciones de Consola, o para enviar comandos a nuestra aplicación que se encuentra ejecutandose ahora mismo.

Documentación Oficial PROCESS

Aquí se encuentra toda la información del módulo:

[Documentacion Oficial](#)

Importación PROCESS

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

Métodos PROCESS

```
console.log('\n P R O C E S S \n');

const { dir } = require('console');
const process = require('process'); //No es necesario, ya que es un objeto global

//console.log(process);

//Devuelve el directorio en el que se encuentra situada la CLI
// Es similar a __dirname, pero la diferencia es que el anterior es
relativo al script o archivo desde donde se llama y .cwd() depende de la
consola
console.log(`\nWorking Directory:\n ${process.cwd()}\n`);

//Devuelve un Arreglo con los parametros que enviamos al momento de
ejecutar nuestra aplicación con:
// node app.js arg1 arg2 ... argn
console.log('\nargv:');
```

```
dir(process.argv);

// Devuelve las Environment Variables del sistema donde nos encontramos
console.log('\nenv:');
dir(process.env);

//Algunas Environment Variables Importantes
console.log('\nenv.LANG:');
dir(process.env.LANG);
console.log('\nenv.TEMP:');
dir(process.env.TEMP);
console.log('\nenv.Path:');
dir(process.env.Path);
console.log('\nenv.USERNAME:');
dir(process.env.USERNAME);

// Devuelve el Process Identifier, este es un numero que el sistema
operativo le asigna a nuestro proceso de Node
console.log('\nprocess.pid:');
console.log(process.pid);

// Entrada de datos por consola
//console.log('\n stdin:');
//console.log(process.stdout.write('holaaaaa\n'));

// Event Listener

// Como dijimos antes, el módulo permite manejar eventos que ocurren
durante la ejecución del proceso o programa
// En este caso el metodo process.on es un event listener y el evento
'exit' se queda esperando a que el proceso haya realizado todas las tareas
para ejecutar una callback

process.on('exit', (code) => {
    console.log(`Codigo de Salida: ${code}`);
    console.log('Saliendo del Programa, estas es la ultima linea');
})

// Metodos para terminar el proceso

// 1. Utilizando un event Listener (Como vimos antes)

//2. process.exit(code): este metodo termina el proceso abruptamente,
permite pasar por parametro un codigo que sera devuelto al sistema para
indicar porqué terminó. Vease la tabla de codigos de salida.

/* process.exit(1); */

// 3. process.kill(pid,signal): Este método envia una señal al proceso que
coincida con el PID que pasamos como argumento, las señales indican
eventos, por lo que el proceso puede realizar logica en base a la señal que
```

```
reciba

/* process.kill(process.pid, 'SIGKILL'); */

// La señal se maneja con process.on(event, callback)

/* process.on('SIGKILL', ()=>{
    console.log('Lo recibí che');
}) */

// 4. process.abort(): Funciona como exit pero la diferencia es que este
devuelve un core file. El core file es un instancia representada en texto
que muestra las variables que se encontraban en memoria hasta el momento de
la finalizacion.

/* process.abort() */

// ESCRIBIR Y LEER EN CONSOLA (P R O C E S S)
// Tenemos que saber que las propiedades que vamos a ver para realizar
estas operaciones devuelven objetos streams y es con esos con los que
operamos al final

// Escribir por Consola
console.log('\nprocess.stdout:');
process.stdout.write('Hola Che');
console.log('\n\n+-----+ \n');

// Leer por consola
process.stdin.on('data', (data)=>{
    process.stdout.write(`process.stdout.write(): ${data}`);

    // Errores por consola
    process.stderr.write('Error: esto es un error por consola :D')
})
```

HTTP MODULE

Antes de entrar de explicar el modulo como tal y sus métodos necesitamos entender y conocer varios conceptos relacionados con redes y protocolos.

Modelo OSI

Debido a la cantidad de fabricantes que existen en el mercado informatico, tanto de software, como de hardware, es que se necesitó crear algun tipo de regla para la comunicación entre los disntintos componentes.

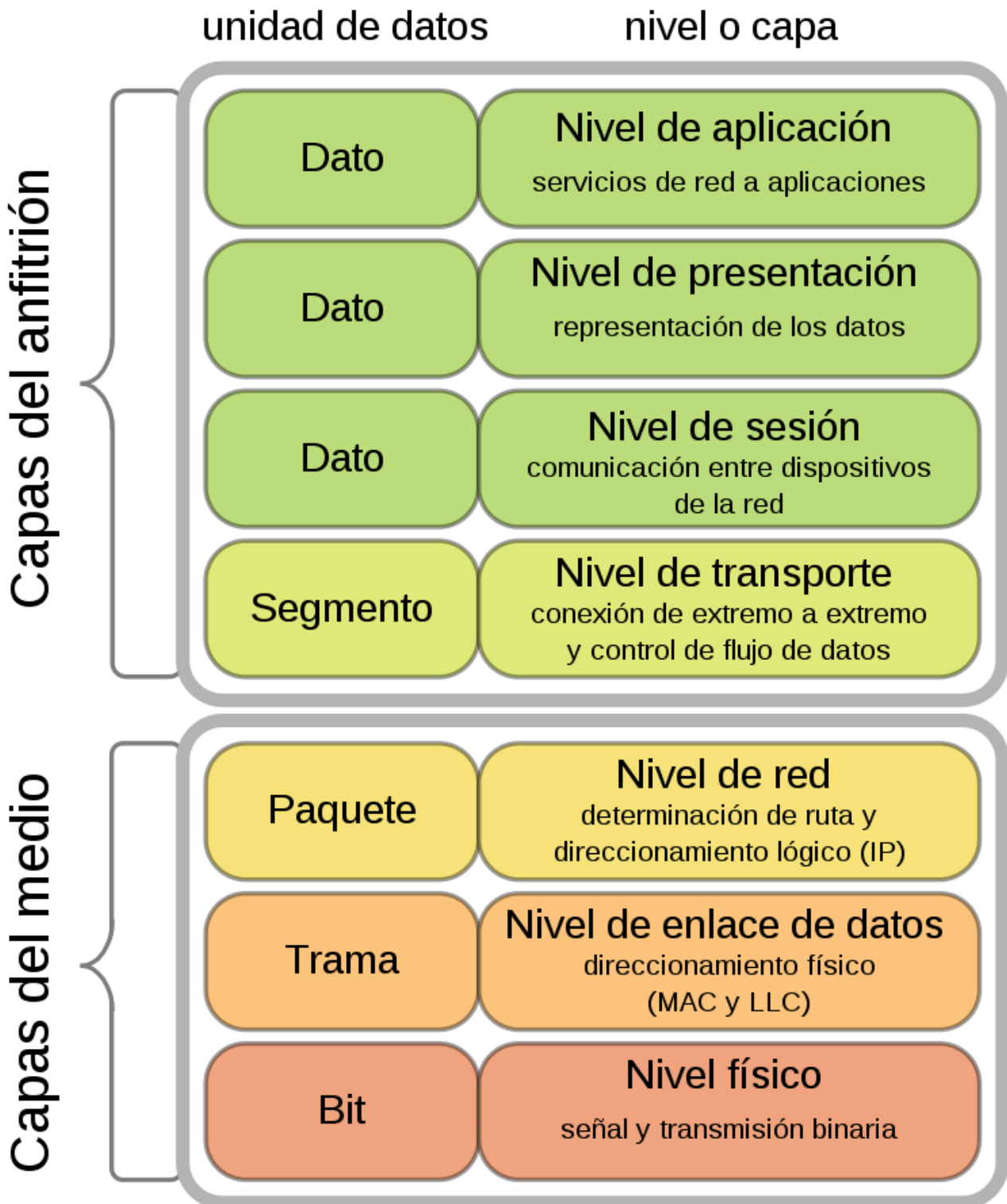
Cuando hablamos de redes informaticas, hoy en dia estas se basan en distintos protocolos para comunicar disntintos tipos de componentes y distintos formas de infomación.

"El modelo de interconexión de sistemas abiertos (ISO/IEC 7498-1), conocido como "modelo OSI", (en inglés, Open Systems Interconnection) **es un modelo de referencia** para los protocolos de la red

(no es una arquitectura de red)"

Como su propia definicion indica, es un modelo de referencia, por lo que, los fabricantes no estan obligados a adoptarlo, aunque hoy en día ya esta siendo utilizado por la mayoria de companias.

Este agrupa protocolos, formas de organizar, llamar y tratar a la información. He aqui un pequeño esquema:

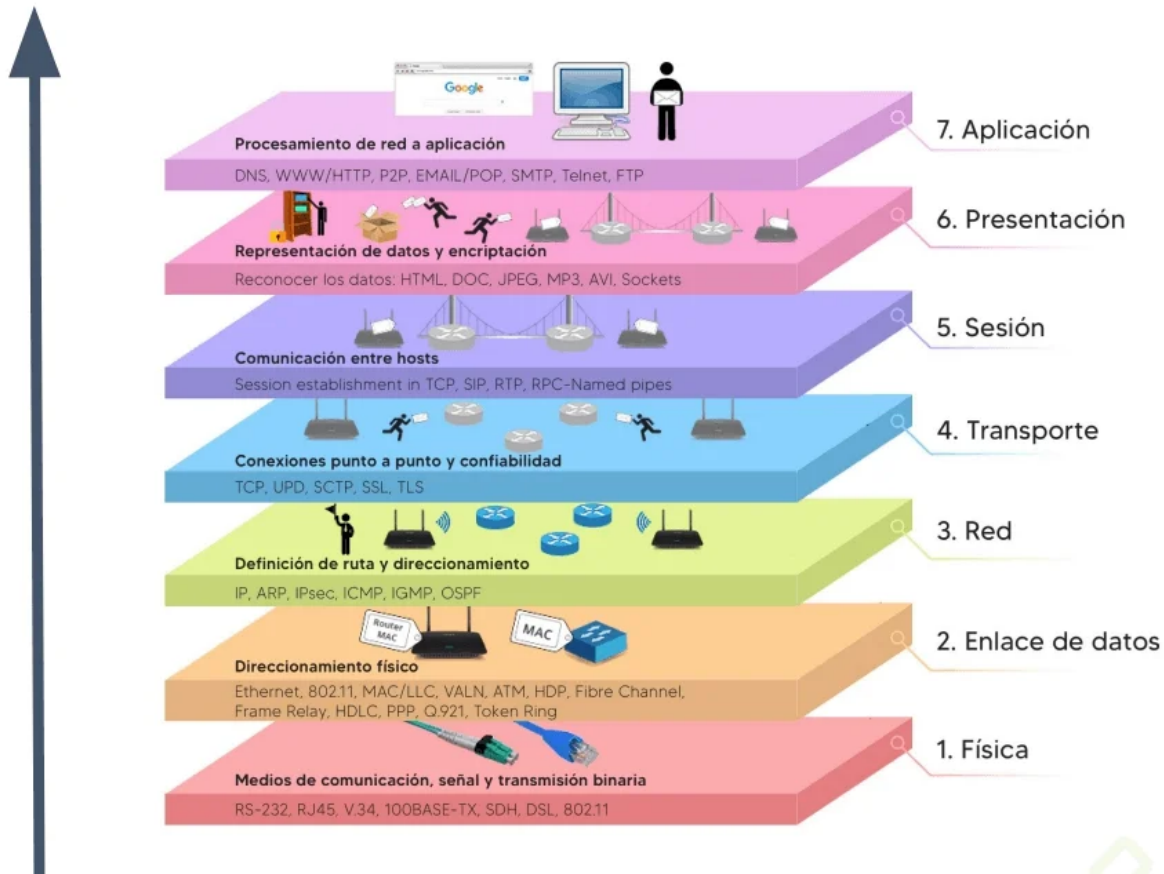


El modelo trabaja de la siguiente manera: Se organiza por capas o niveles, de los cuales el primero es el de mayor complejidad y el primero es mas sencillo o parecido a nuestro lenguaje. Cada nivel tiene su propia

forma de tratar y nombrar a la información con la trabaja (Unidad de datos).

Cada nivel trabaja con sus propios protocolos y dispositivos de red. Por ejemplo en el nivel físico, aparecen los bits, las señales electricas, los cables de ethernet, y cualquier otro medio que se utilizó para el envío de datos binarios.

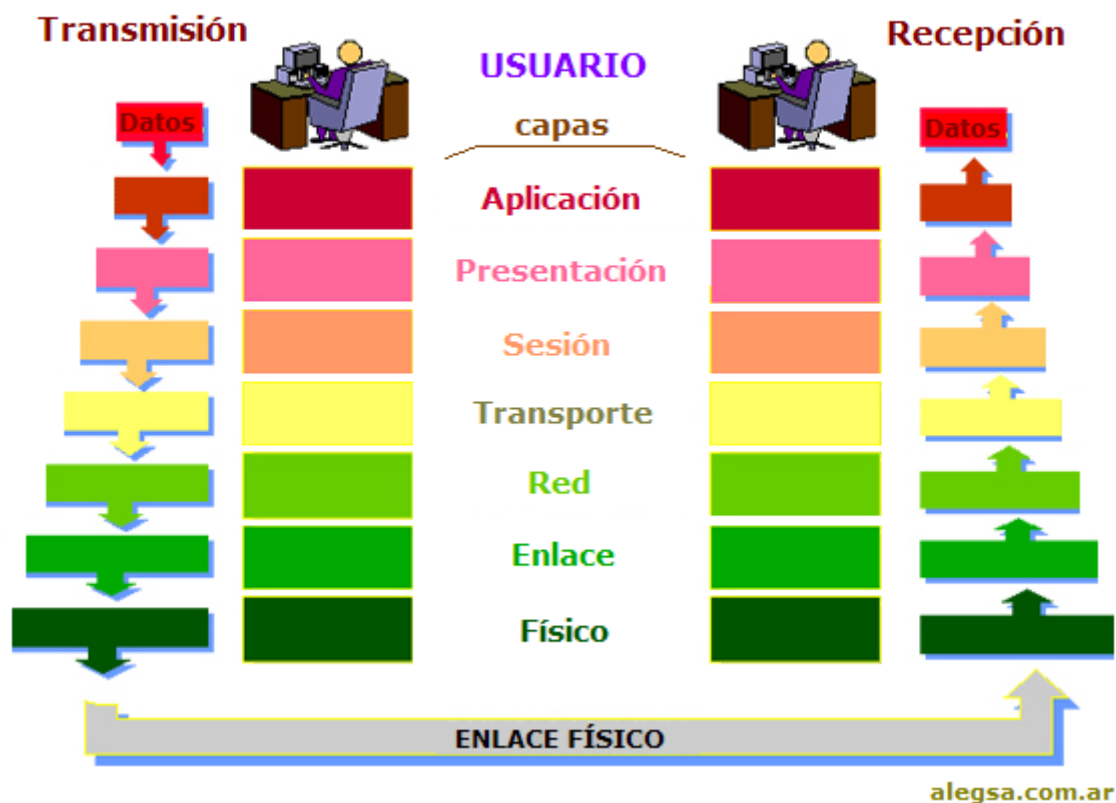
Mientras que en el nivel de aplicación, como su nombre lo indica, aparecen los protocolos que usamos comunmente



Platzi

Con respecto a los paquetes de información, podemos decir que este sistema es similar a los sistemas de envíos de cartas de la **vida real**. En el que cada nivel añade su parte de información que debiera ser tratada en el nivel par, esto quiere decir que, si nivel de red del TRANSMISOR añade al paquete, la IP del destinatario, entonces cuando el paquete llegué al RECEPTOR su nivel de red debiera realizar las operaciones pertinentes para verificar que esa IP se corresponde con la de la maquina actual.

Las 7 capas del modelo OSI



Mas información: [Mas Informacion](#)

El Protocolo TCP/IP

El modelo TCP/IP es usado para comunicaciones en redes y, como todo protocolo, describe un conjunto de guías generales de operación para permitir que un equipo pueda comunicarse en una red. TCP/IP provee conectividad de extremo a extremo especificando cómo los datos deberían ser formateados, direccionados, transmitidos, enrutados y recibidos por el destinatario.

TCP/IP está compuesto por dos protocolos principales: el Protocolo de Control de Transmisión (TCP) y el Protocolo de Internet (IP).

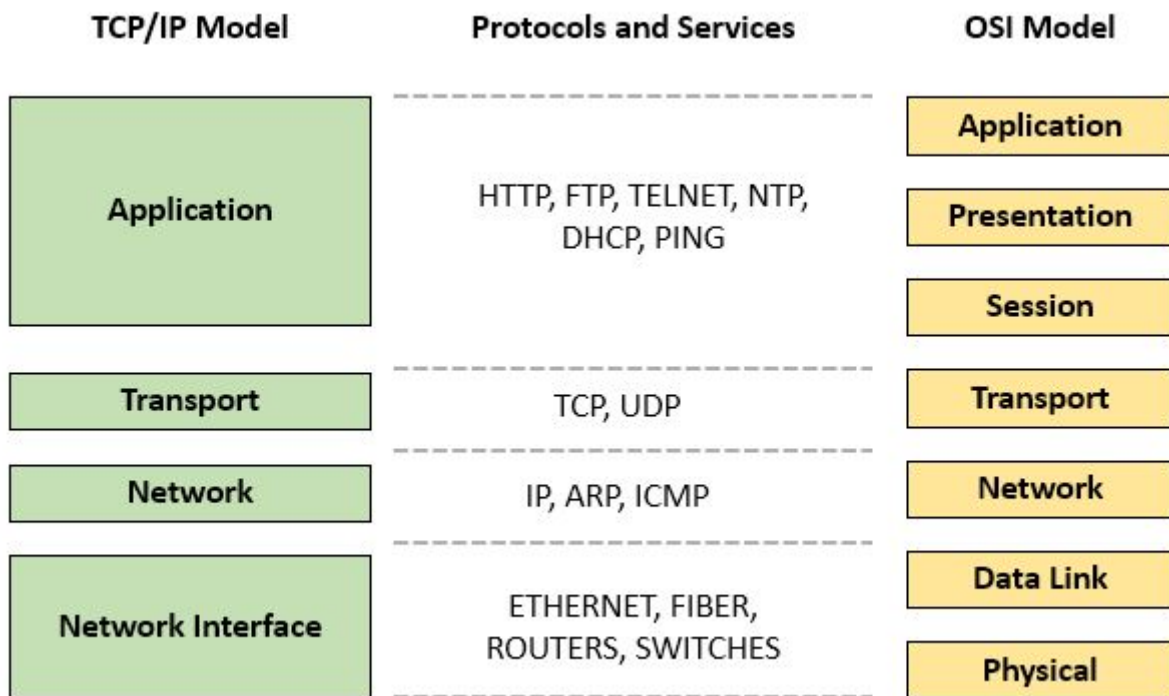
1. TCP/IP: Es el encargado de controlar la integridad y el orden en el que llegan los datos, fragmenta el mensaje en pequeños paquetes y si algún paquete se pierde en la transmisión, el TCP lo detecta y lo vuelve a enviar.
2. IP: Es el encargado de enviar los paquetes a través de la red, determina la ruta, y asegura que el paquete llegue al destino.

Otros protocolos que forman parte de TCP/IP son:

- DNS(Domain Name System): Que permite traducir los nombre de dominios alojados en los distintos servidores DNS. Los nombres de dominios son direcciones de texto que se corresponden a una direccion IP en especifico. Por ejemplo *www.google.com = 64.233.191.255*, la primera es el nombre de dominio y la segunda la dirección IP.

- **FTP: (File Transfer Protocol):** Como su nombre lo indica permite transferir archivos, algo a destacar es que este protocolo no se encuentra cifrado, por lo que no es seguro. En cambio podemos utilizar SFTP (Secure File Transfer Protocol) o FTPS (File Transfer Protocol sobre SSL/TLS).
- **UDP: (User Datagram Protocol):** Este protocolo sería el opuesto al TCP, no controla la integridad ni el flujo de los paquetes de datos, al no verificar la integridad de los paquetes que llegan ni el orden, no podemos confiar que los datos lleguen de la forma correcta, pero esto lo compensa con su alta velocidad a la hora de realizar la transmisión de los paquetes.

El modelo TCP/IP también se divide por capas o niveles, al igual que el modelo OSI. Existe una relación entre las capas del modelo OSI y el TCP/IP, ya que comparten protocolos.



En el caso del TCP/IP, las capas son:

1. **Interfaz de Red o Acceso:** Esta capa establece el tipo de conexión entre 2 dispositivos, si la conexión será por Ethernet, WiFi o PPP.
2. **Red:** Se encarga de realizar conexión entre redes diferentes y además es la que verifica que el paquete llegó a destino. Aquí aparece el protocolo IP.
3. **Transporte:** Es la que controla el flujo, detección, corrección de errores, segmentación y reensamblado de los paquetes. Aquí aparece TCP y UDP.
4. **Aplicación:** En esta capa se define la estructura del mensaje a enviar, los servicios a utilizar y la forma de transmisión. En ella se incluyen HTTP (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol) y FTP (File Transfer Protocol).

Ahora, una vez que conocemos las "reglas" de la web e internet, estamos habilitados para hablar del protocolo que nos va a permitir realizar las operaciones esenciales que forman parte de las páginas web.

El Protocolo HTTP

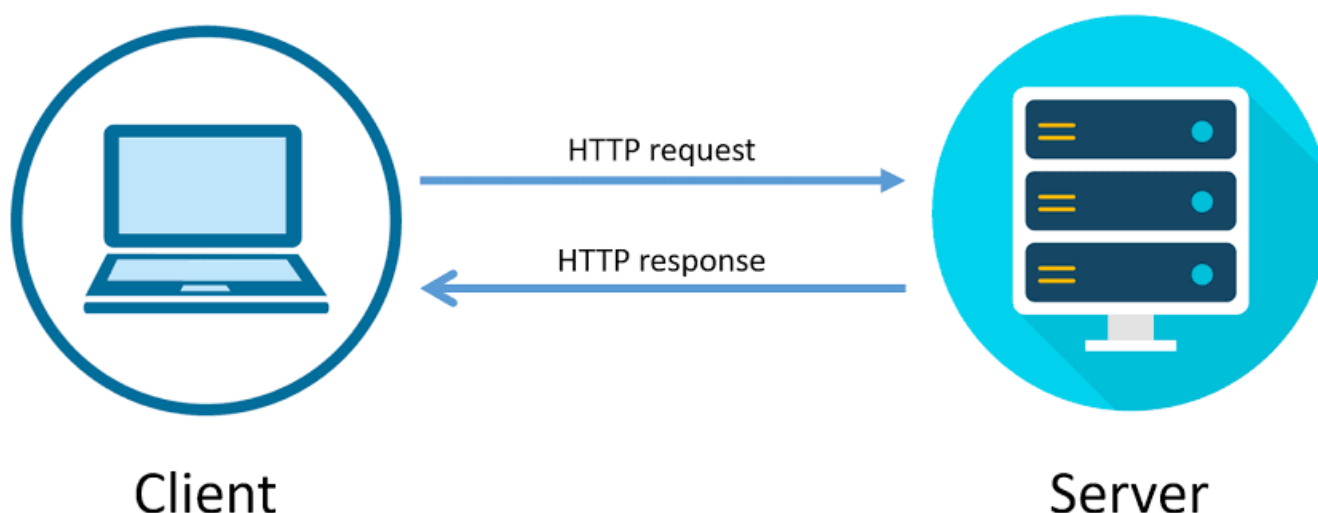
Páginas Interesantes: [MDN](#), [WIKIPEDIA](#)

Como indicamos antes, este protocolo forma parte de la capa de aplicación (tanto de OSI como de TCP/IP), y como mencionamos en esta se nos habilita a estructurar la forma de los mensajes que enviamos. Pues bien, este protocolo lo que nos permite es realizar la transmisión de documentos hipermedia o hipertexto.

Hipermedia o hipertexto es una estructura **no secuencial** que permite crear, agregar, **enlazar** y compartir información de **diversas fuentes** por medio de **enlaces** asociativos y redes sociales. La forma más habitual de hipertexto en informática es la de hipervínculos o referencias cruzadas automáticas que van a otros documentos (lexías).

Basandonos en la definición anterior podemos darnos cuenta, que nos referimos a archivos HTML y XML. Ahora bien, no necesariamente tienen que ser este tipo de documentos, también podemos enviar imágenes, audio, y otros.

El protocolo HTTP se basa en el **MODELO CLIENTE-SERVIDOR**, este no es más ni menos que la forma en la cual se comunican los dispositivos. Normalmente cuando lo utilizamos realizamos peticiones (HTTP REQUESTS) a un servidor web, el cual procesa la petición y nos devuelve una respuesta (HTTP RESPONSE). Algo importante a marcar es que en este modelo, **SIEMPRE** la comunicación la inicia el cliente. Mas adelante presentaremos otro método que nos permitiera realizar peticiones en ambas direcciones.



Las 2 entidades que se presentan en este modelo son: el cliente, que lo llamaremos Agente de Usuario, y el Servidor Web.

- El Agente de Usuario: Es cualquier herramienta o interfaz que le permita al usuario realizar las peticiones al servidor, por lo general este es el **navegador**, aunque bien podría ser, por ejemplo, **Postman**. Si utilizamos un navegador, este se encargará de realizar varias peticiones para traer todos los elementos necesarios para mostrar o utilizar para el recurso al que estamos accediendo (La Página Web).
- El Servidor Web: Como dijimos, es aquel que procesa las peticiones y el que se encarga de enviar los datos al Agente de Usuario.

Características de HTTP

- Sencillez: Conceptualmente y Estructuralmente es un protocolo simple de procesar y de entender. Por lo que no tiene muchos requisitos para ser utilizado, ya sea, por distintos agentes o aplicaciones.
- Extensible: Gracias a su estructura este puede ser facilmente utilizado para distintos propositos y permite **añadir nuevas funcionalidades**.
- Existen Sesiones pero No Estados: Esto quiere decir que entre las distintas peticiones de un misma sesion, HTTP no guarda información, un ejemplo sencillo es el caso de los carritos de compras en los e-commerce. Pero si permite crear dichas sesiones para dar un cierto contexto a las páginas (Cookies).
- Conexiones: Ya que es un protocolo de la capa de aplicación, este puede hacer uso de TCP, para controlar el flujo de datos y para la deteccion de errores en los mensajes.

Como trabaja HTTP

A continuación vamos a describir el modo en el que opera HTTP.

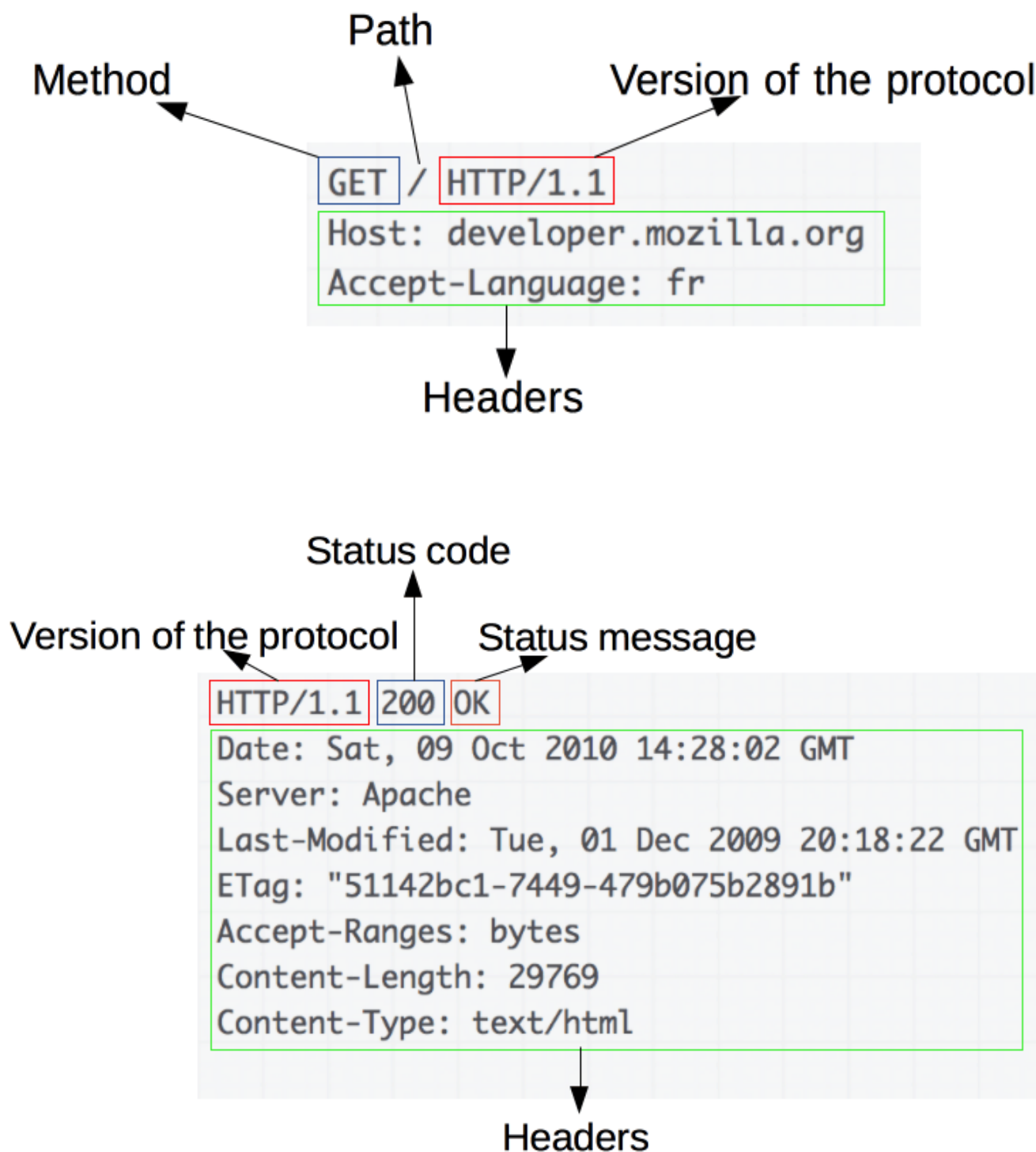
1. Usamos TCP para establecer o reutilizar conexiones existentes con el Servidor con el se comunicará
2. Realizar la petición HTTP (Request)
3. Esperar la respuesta del servidor (Response)
4. Se cierra la conexión TCP o se rehusa

Estructura HTTP

```
GET / HTTP/1.1 Host: developer.mozilla.org Accept-Language: fr
```

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

```
<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
```



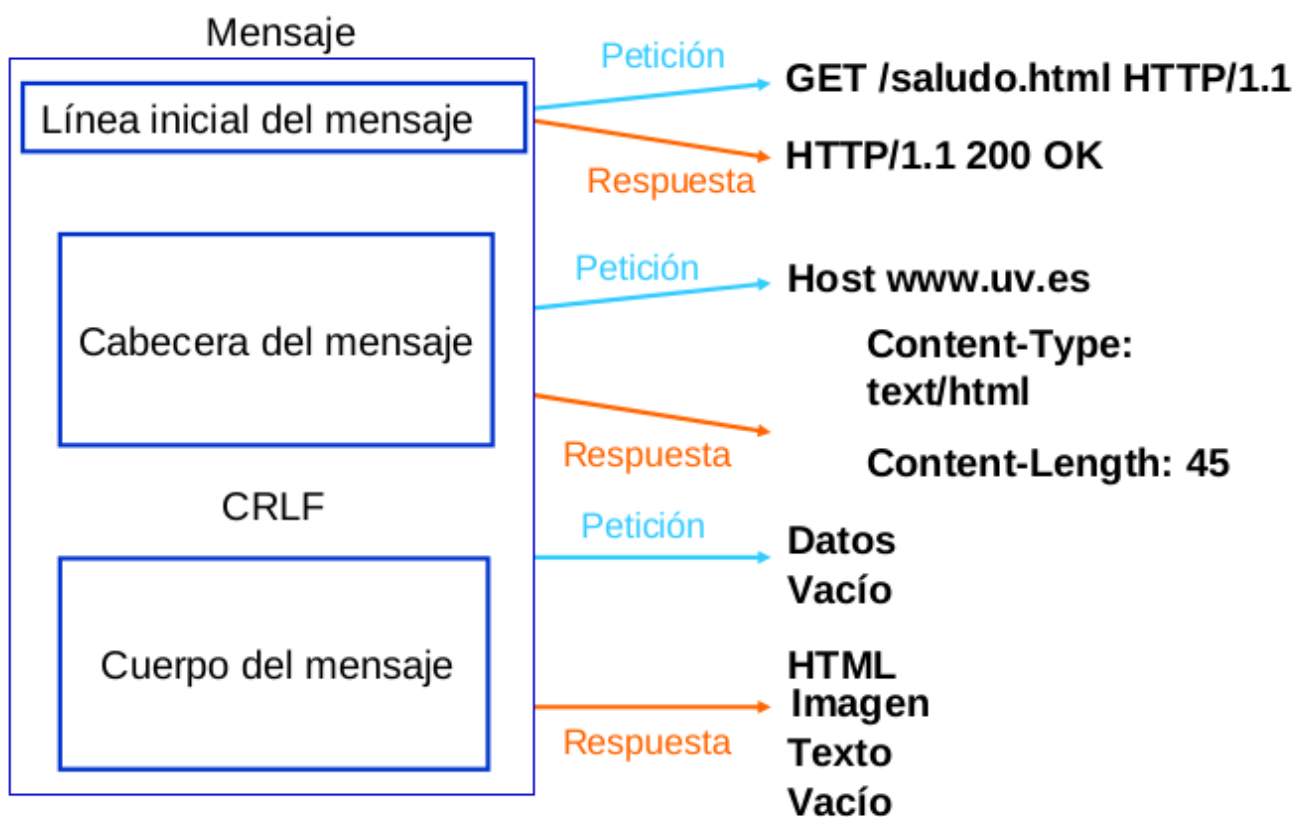
Las Peticiones (Requests) y Respuestas (Response) de HTTP tienen una estructura que se divide en:

Primera Línea: Es la que lleva la información del protocolo y la información fundamental para realizar la operación, en el caso de que sea una petición se aclara la versión de HTTP, el METODO HTTP y el recurso al que se quiere acceder (Ruta). Y en el caso de las respuestas se muestra la versión de HTTP y el CODIGO de la RESPUESTA.

Encabezados/Cabeceras: Aquí se guardan los metadatos del mensaje, cada cabecera tiene un nombre y un valor asignado, al ser una estructura tan simple esta puede mutar y pueden aparecer nuevos datos. Algunos de los datos que pueden aparecer son:

- Cabeceras que indican las capacidades aceptadas por el que envía el mensaje: Accept (indica el MIME aceptado), Accept-Charset (indica el código de caracteres aceptado), Accept-Encoding (indica el método de compresión aceptado), Accept-Language (indica el idioma aceptado), User-Agent (para describir al cliente), Server (indica el tipo de servidor), Allow (métodos permitidos para el recurso)
- Cabeceras que describen el contenido: Content-Type (indica el MIME del contenido), Content-Length (longitud del mensaje), Content-Range, Content-Encoding, Content-Language, Content-Location.
- Cabeceras que hacen referencias a URIs: Location (indica donde está el contenido), Referer (Indica el origen de la petición).
- Cabeceras que permiten ahorrar transmisiones: Date (fecha de creación), If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, If-Range, Expires, Last-Modified, Cache-Control, Via, Pragma, Etag, Age, Retry-After.
- Cabeceras para control de cookies: Set-Cookie, Cookie
- Cabeceras para autenticación: Authorization, WW-Authenticate
- Cabeceras para describir la comunicación: Host (indica máquina destino del mensaje), Connection (indica como establecer la conexión) Otras: Range (para descargar solo partes del recurso), Max-Forward (límite de cabeceras añadidas en TRACE).

Cuerpo/Body: Esto representa la respuesta como tal por parte del servidor, al cliente. Aquí pueden aparecer tanto HTML, XML, JSON, PLAINTEXTS como archivos MULTIMEDIA.

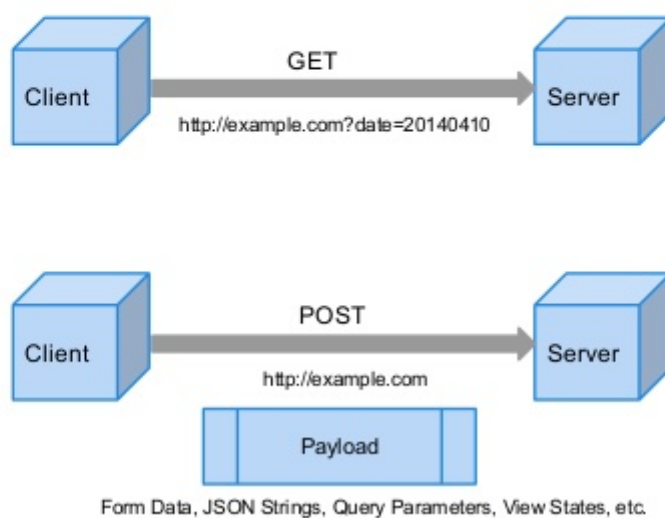


Métodos HTTP

Los métodos HTTP indican el tipo de acciones y las características de la petición HTTP. Debido a las características mencionadas anteriormente, la cantidad de métodos también han aumentado a lo largo del tiempo. Pero ahora vamos a presentar los más importantes:

- GET: Pide un recurso en específico, solo recupera datos.
- POST: Envía datos para ser procesados por un recurso (un recurso puede ser un archivo o programa corriendo en el servidor), además los datos se incluyen en el cuerpo de la petición y no en la URL (Como si ocurre con GET).

GET vs. POST



- PUT: También envía datos, pero no se hace referencia al recurso que lo procesará en la URL. Además está orientado a actualizar datos ya existentes, a diferencia de POST, que está orientado a crear nuevos datos.
- DELETE: Como su nombre lo indica, borra un recurso especificado por la URL.
- OPTIONS: Espera los métodos HTTP que soporta la URL que indicamos
- HEAD: Funciona como un GET, pero omite el cuerpo, solo trae las cabeceras (Sirve para "simular" un GET).

Hay muchos más métodos pero estos son los más importantes, si desea verificar o conocer los demás, visite la documentación oficial:

[Documentación Oficial](#)



Tecsify presenta...

www.Tecsify.com

Métodos de petición HTTP



El protocolo HTTP regula la forma en la que el cliente realiza peticiones y la forma en la que responde el servidor, para esto emplea diferentes métodos de petición

A continuación, te contaremos más detalles acerca de estos métodos.



GET:

El método GET solicita una un recurso específico.

Las peticiones que usan el método GET solo deben recuperar datos.



HEAD:

Este método pide una respuesta idéntica a la de una petición GET, pero sin el cuerpo de la respuesta, únicamente con el encabezado de la solicitud.



POST:

Se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.



PUT:

Es similar al POST, solo que el método PUT se utiliza para la actualización de información existente, es semejante a un UPDATE a nivel de base de datos.



DELETE:

Permite eliminar un recurso específico, generalmente se utiliza para eliminar información existente, es semejante a un DELETE a nivel de base de datos.



PATCH

Este método se emplea generalmente para realizar modificaciones parciales de un recurso en particular.

Los métodos **PUT & DELETE** son **idempotentes**; es decir, puede ser ejecutados **varias veces** y tener el mismo efecto, caso contrario a un **POST** que cada vez que se ejecuta realiza la agregación de un **nuevo objeto**.

Algunos métodos solo pueden aplicarse en ciertos contextos, por ejemplo, el método **CONNECT**, que crea una conexión directa y protegida por medio de un proxy (tunneling)

[/Tecsify](https://www.facebook.com/Tecsify)[@Tecsify](https://www.instagram.com/Tecsify)[@Tecsify](https://twitter.com/Tecsify)www.Tecsify.com/blog[/Tecsify](https://www.youtube.com/Tecsify)[Tecsify](https://www.linkedin.com/company/Tecsify)[@Tecsify](https://www.tiktok.com/@Tecsify)

GET

`/pet/{petId}` Find pet by ID

PUT

`/pet` Update an existing pet

DELETE

`/pet/{petId}` Deletes a pet

POST

`/pet/{petId}/uploadImage` uploads an image

Códigos de respuestas HTTP

Versiones de HTTP

Video Interesante: [HTTP/1 to HTTP/2 to HTTP/3 | ByteByteGo](#)

Documentación Oficial HTTP

Aquí se encuentra toda la información del módulo:

Documentacion Oficial

Importación HTTP

Metodos HTTP