**Node.js**

Como ya vimos en el capítulo de NPM, Node.js es una librería y entorno de ejecución de Entrada/Salida (o I/O), dirigida hacia eventos. Node es asíncrono y a diferencia de otros lenguajes de programación, se basa en un solo hilo para ejecutar todas sus operaciones. Node.js está basado en el motor V8 de Google, en el cual también se basa el navegador Google Chrome.

Node.js fue concebido como un entorno de ejecución de JavaScript orientado a eventos de asíncronos. Node.js está diseñado para posibilitar la construcción de aplicaciones escalables.

Esto contrasta con el modelo de concurrencia más común hoy en día, donde se usan hilos del Sistema Operativo. Las operaciones de redes basadas en hilos son relativamente ineficientes y son muy difíciles de usar. Además, los usuarios de Node están libres de preocupaciones sobre el bloqueo del proceso, ya que no existe. Casi ninguna función en Node realiza I/O (Input/Output [Entrada/Salida]) directamente, así que el proceso nunca se bloquea. Debido a que no hay bloqueo es muy razonable desarrollar sistemas escalables en Node.

Node tiene un diseño similar y está influenciado por sistemas como Event Machine de Ruby ó Twisted de Python. Node lleva el modelo de eventos un poco más allá, este presenta un bucle de eventos como un entorno en vez de una librería. En otros sistemas siempre existe una llamada que bloquea para iniciar el bucle de eventos. El comportamiento es típicamente definido a través de callbacks al inicio del script y al final se inicia el servidor mediante una llamada de bloqueo como EventMachine::run(). En Node no existe esta llamada. Node simplemente ingresa el bucle de eventos después de ejecutar el script de entrada. Node sale del bucle de eventos cuando no hay más callbacks que ejecutar. Se comporta de una forma similar a JavaScript en el navegador - el bucle de eventos está oculto al usuario.

HTTP tiene un módulo, el cual es tratado con imporante en Node, diseñado con operaciones de streaming y baja latencia en mente. Esto hace a Node candidato para ser la base de una librería o un framework web.

Solo porque Node esté diseñado sin hilos, no significa que no se pueda aprovechar los múltiples cores de su sistema. Procesos hijos pueden ser lanzados usando la API child\_process.fork(), la cual está diseñada para comunicarse fácilmente con el proceso principal.

**Hola Mundo en Node**

En el siguiente ejemplo, se mostrará el ya conocido “Hola Mundo”, en el que se pueden manejar muchas conexiones concurrentes.

Concebido como un entorno de ejecución de JavaScript orientado a eventos asíncronos, Node está diseñado para construir aplicaciones en red escalables. En la siguiente aplicación de ejemplo "hola mundo", se pueden manejar muchas conexiones concurrentes de diferentes clientes con un solo hilo, gracias a la manera en la que se construyó Node.

**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('Hola Mundo\n');

});

server.listen(port, hostname, () **=>** {

console.log(`El servidor se está ejecutando en http://${hostname}:${port}/`);

});

Las anteriores 11 líneas son más que suficientes para poner a funcionar un servidor de Node.

Primero que nada, para poder manejar el protocolo http en una aplicación de Node en blanco, lo que tenemos que hacer es importar el módulo ‘http’ que ya viene incluido por defecto en Node, no hay necesidad de instalar ningún paquete adicional.

Lo que hace ***require*** es “requerir” o importar código que esté en los módulos integrados en el núcleo de Node.js, importar código que esté en *node\_modules,* o incluso, importar código que nosotros mismos hayamos creado anteriormente. En este caso, sólo importamos el módulo ***http*** del núcleo de Node.js

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

Después, tenemos que declarar la dirección IP de nuestro host. En este caso, es la IP de *localhost*

**const** hostname = '127.0.0.1';

Al igual que la dirección IP, también tenemos que indicarle a Node por cuál puerto tiene que escuchar las peticiones.

**const** port = 3000;

A continuación, podemos ver cómo se utiliza el módulo ‘http’, con la función *createServer.* Esta función acepta como parámetro una *callback,* la cual será ejecutada cada vez que un cliente haga una petición al servidor. Una “callback” resumidamente es una función anónima que se ejecutará una vez que la función createServer se termine de ejecutar, esto es debido a la naturaleza de Node y de Javascript de correr en un solo hilo, también llamado “single thread”.

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

res.statusCode = 200;

res.setHeader('Content-Type', 'text/plain');

res.end('Hola Mundo\n');

});

Las variables “req” y “res” son inyectadas por la función *createServer* al finalizar su proceso.

Una manera de verlo sería la siguiente:

**function** createServer(nuestraFuncionAnonimaComoParametro) {

*//Todo el código necesario del módulo 'http'...*

...

*//Sin embargo, al final de toda la cadena de callbacks*

*//requerida para hacer funcionar un servidor, 'createServer'*

*//llama a nuestra función, pasándole como parámetros 'req'*

*//y 'res'*

nuestraFuncionAnonimaComoParametro( req, res );

}

Aunque realmente el módulo *http* de Node.js hace mucho más que sólo eso, la anterior representación simplifica la manera de explicar cómo funciona una *callback* en Node.js. Es realmente importante entender este patrón, ya que es una de las bases de cómo función Node.

Siguiendo con el ejemplo:

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

*//En este momento las variables ‘req’ y ‘res’ ya son accesibles, ya que*

*//la función ‘createServer’ llamó a* nuestraFuncionAnonimaComoParametro

*//pasándole ‘req’ y ‘res’ cómo parámetros.*

*//El código del estado de la petición.*

res.statusCode = 200;

*//El tipo de contenido que se enviará de vuelta al cliente.*

res.setHeader('Content-Type', 'text/plain');

*//Finalmente enviamos el contenido al cliente con ‘res.end’.*

res.end('Hola Mundo\n');

});

Los objetos *req* y *res* son objetos realmente importantes en cuanto a servidores http en Node se refiere.

***req*** es un objeto que contiene información acerca de una petición (request) hecha por un cliente; por ejemplo, un navegador web como Firefox. Por otro lado, ***res*** es la respuesta (response) usada para devolver los valores deseados al cliente que hizo la petición; por ejemplo, devolver las ventas del mes de enero de alguna tienda, o en nuestro caso, una *string* con el valor de “Hola Mundo”.

Los objetos *‘req’* y *‘res’* pueden ser nombrados de cualquiera manera. Es decir:

//Esto sería lo mismo

(req, res) **=>** {

}

//Que esto

(peticion, respuesta) **=>** {

}

Sin embargo, por convención, se recomienda nombrarlas ‘req’ y ‘res’.

Por último, gracias a que guardamos referencia al objet que retorna la función *createServer* en una constante **const** server = http.createServer, podemos hacer referencia a dicho objeto y llamar a una de sus funciones llamada “listen”. Esto le indica al Node en qué puerto de nuestra computadora escuchar a las peticiones entrantes (3000 en nuestro caso) y por el host que hayamos definido (127.0.0.1 en nuestro caso).

server.listen(port, hostname, () **=>** {

*//Una vez que nuestro servidor esté escuchando, se imprimirá el contenido del*

*//siguiente console.log en la consola.*

console.log(`El servidor se está ejecutando en http://${hostname}:${port}/`);

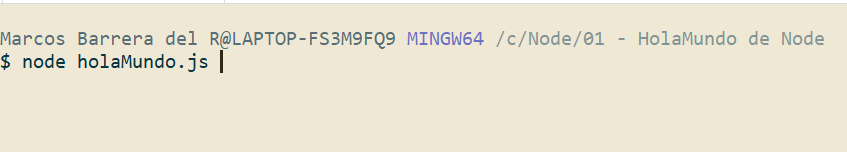
});

**NOTA:** A manera de recordatorio, *console.log* nos permite imprimir en consola los valores de nuestras variables, para poder depurar errores en JavaScript y por ende, en Node.js.

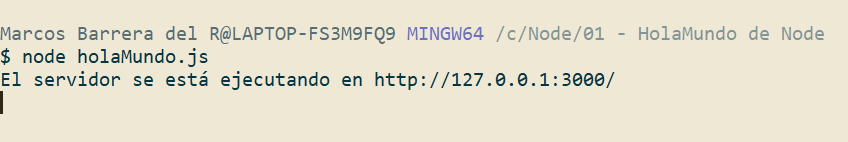
Perfecto, pero falta algo… ¿Cómo hacemos para poner a funcionar nuestro servidor? Sencillo, lo que tenemos que hacer es crear un archivo con el código que explicamos anteriormente en nuestro editor de preferencia.



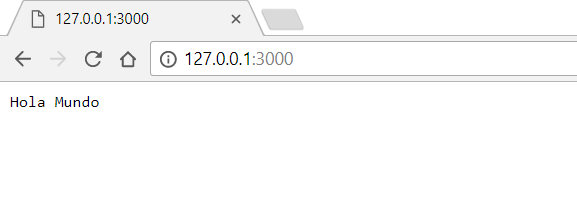
Una vez escrito el código, lo guardamos en el directorio que queramos y desde la terminal, nos posicionamos en dicho directorio. Una vez ahí, para poder ejecutar nuestro servidor, escribiremos lo siguiente:



Después daremos Enter y nos aparecerá algo similar a esto:



Y así de sencillo, nuestro servidor estará activo y corriendo en Node. Para comprobarlo, nos dirigiremos a nuestro navegador de preferencia y escribiremos en la barra de URL *http://127.0.0.1:3000/*

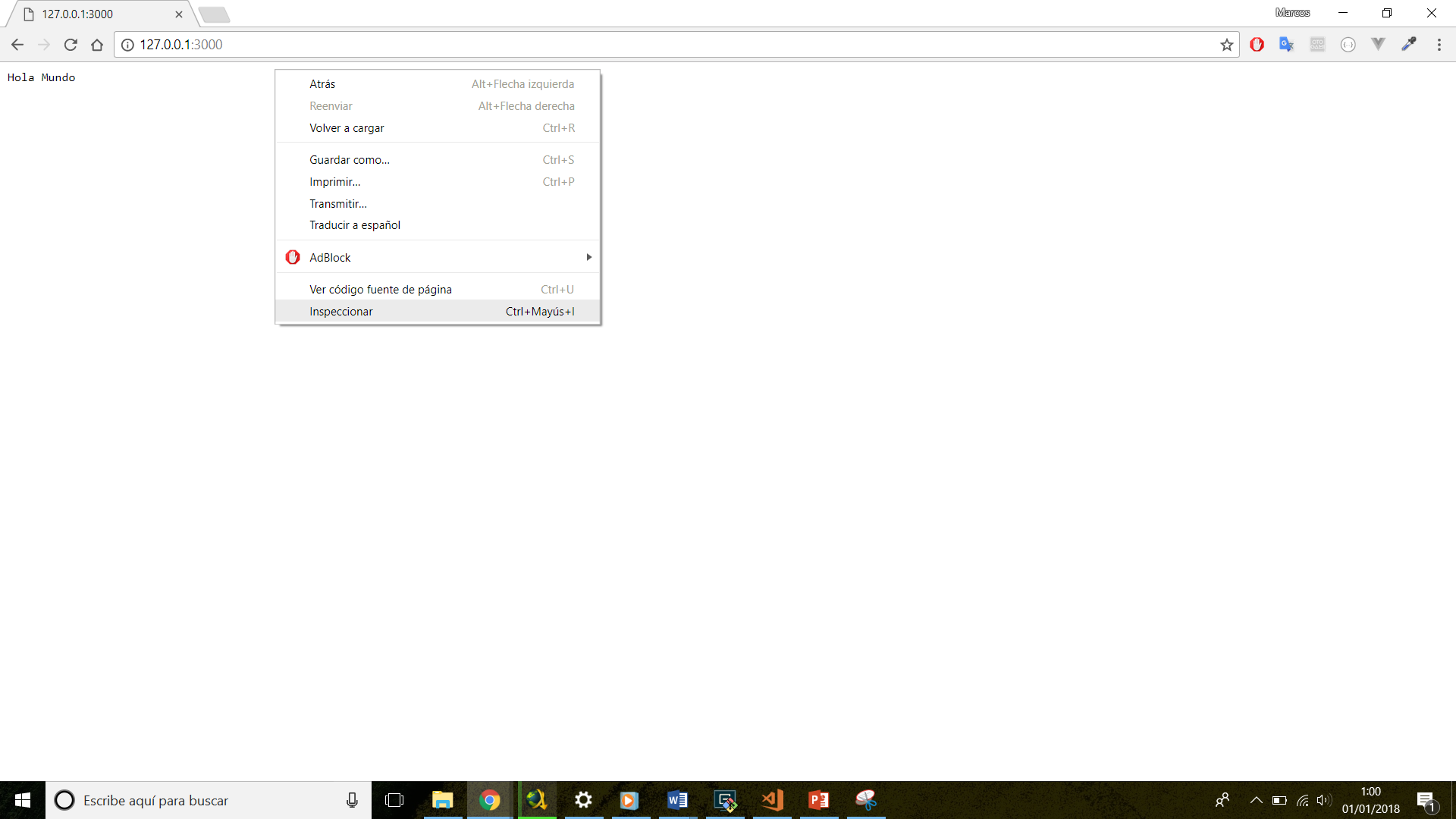


Al hacer esto, podemos observar como el contenido que escribimos anteriormente es mandado a nuestro navegador. Hablamos de esta línea de código:

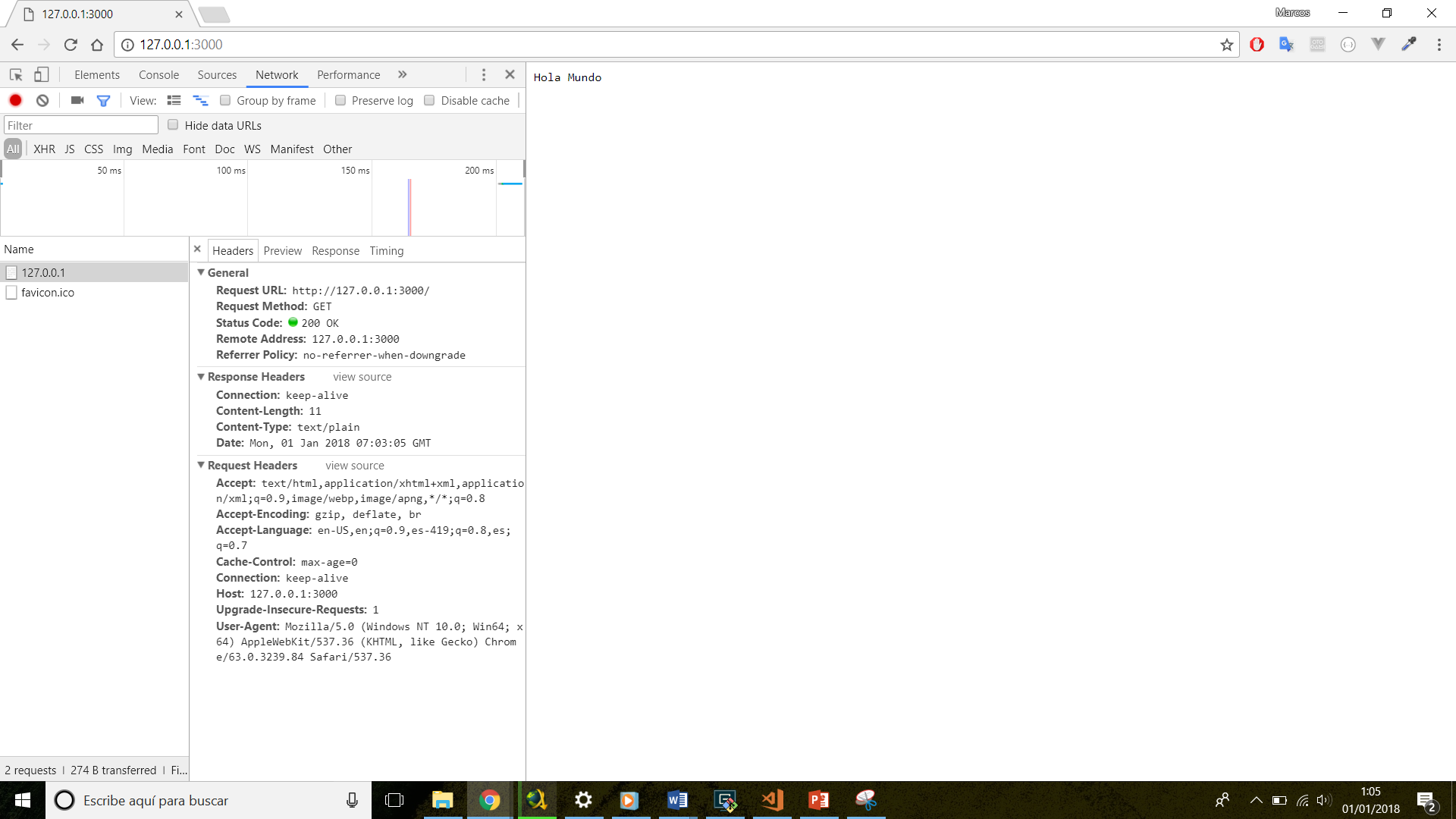
res.end('Hola Mundo\n');

Entonces, recapitulando, lo que pasó realmente, es que configuramos nuestro servidor en el puerto 3000 con el host 127.0.0.1, Node hizo todo el trabajo pesado en su funcionalidad interna y también le indicamos que por cada petición de un cliente (como nuestro navegador web), les respondiera con un statusCode de 200, con el header de tipo de contenido de texto plano y que al final le enviará al cliente el texto plano de “Hola Mundo”. Así que después de ejecutar la línea de “node holaMundo.js”, Node procedió a escuchar peticiones de cualquier cliente y en el momento en el que ingresamos “127.0.0.1:3000” en nuestro navegador y dimos Enter. Nuestro navegador hizo una petición HTTP y Node respondió con lo que establecimos en código.

Si damos clic derecho e Inspeccionar en Google Chrome…



Después en la pestaña de *Network,* y por último en la petición al servidor “127.0.0.1”, podremos observar lo siguiente:



Si miramos detenidamente, podremos ver que el Status Code tiene el valor de **200**, tal como lo establecimos en código desde nuestra aplicación de Node.js. Y más abajo, podemos observar cómo el Content Type tiene el valor de **text/plain**. Tal como lo definimos en nuestro código.

Muy bien, después de haber explicado cómo crear un simple servidor y de lo sencillo que es, debemos de ver cómo Node hace posible esto y en qué se basa. Lo cual nos lleva al siguiente tema: El motor V8.

**V8**

Entonces… ¿Qué es V8?

Según la propia página oficial, “V8 es un proyecto de Google *open source* (código abierto). V8 es un motor de JavaScript de alta eficiencia escrito en C++. Es usado en el navegador Google Chrome, Node.js, entre otros. Implementa ECMAScript (una especificación de lenguaje de programación para JavaScript), y puede ser ejecutado en Window 7 o mayor, macOS 10.12 o mayor y en sistemas Linux que usen procesadores compatibles. **V8 puede ser usado como una aplicación independiente o puede ser incrustado en cualquier aplicación de C++**”.

Es importante que tengas en cuenta el texto en negritas, porque luego retomaremos ese tema. Sólo recuerda, V8 puede ser incrustado y utilizado en otra aplicación de C++. Aunque nosotros no trabajaremos directamente con C++, Node internamente sí que lo hace.

¿Pero por qué es tan importante V8 para Node.js? Bueno… sin V8, Node no podría comunicarse con las partes de más bajo nivel de nuestro sistema, como tarjeta RAM, procesador, disco duro, etcétera. La siguiente imagen que hicimos podría clarificar un poco cómo encaja V8 en el esquema de Node.js



Puede que pienses que esto es irrelevante, pero para tener un buen entendimiento de lo que es Node y cómo funciona, es necesario explicarlo. Como podemos ver en la anterior ilustración, V8 trabaja en un nivel intermedio, ya que está escrito en C++, ¿pero para qué nos sirve que V8 esté escrito en C++ si lo que estamos aprendiendo en Node.js, el cual está basado en JavaScript? Buena pregunta, la respuesta a esa pregunta viene a continuación.

Para que JavaScript pueda ser ejecutado, necesita de un motor en el cual poder correr. De hecho, al igual que V8 es usado en el navegador web Google Chrome, Firefox corre en un motor llamado Gecko, Safari corre en un motor conocido como WebKit, Microsoft Edge está basado en un motor Chakra… ¿ves el patrón? Esto significa que para que JavaScript funcione, tiene que tener un motor internamente.

En este caso, V8 está incrustado en Node.js, esto le permite a Node tener la capacidad de comunicarse con el hardware de la computadora gracias al poder de C++, sin embargo, nosotros no escribimos nuestro código en C++, sino que lo escribimos en JavaScript. Esto saca partido de lo bueno de los dos mundos. Es decir, desde Node podemos escribir código de JavaScript que nos permita leer archivos mediante buffers, nos permite hacer uso de Streams, y como vimos anteriormente, podemos incluso manejar el protocolo HTTP para construir servidores web, entre muchas otras funcionalidades. Todo esto por sí solo, JavaScript no lo soporta ni está estandarizado en las especificaciones de ECMAScript, pero gracias a que en una capa más baja está corriendo V8, es posible utilizar las características que un lenguaje de nivel medio como C++ tiene desde JavaScript. En pocas palabras, la mayoría de funcionalidad que pueda realizar C++ puede ser traspasada a JavaScript con la ayuda de V8.

Entonces, en resumen, Node.js usa JavaScript que corre en V8, este último hace de intérprete o traductor entre el código de JavaScript y C++, luego V8 traduce ese código a Lenguaje Ensamblador, después el código se interpreta en Lenguaje Maquina y por último el Lenguaje Maquina se comunica con el hardware.

Sin embargo, hay un importante detalle, Node.js es asíncrono y V8 es síncrono… ¿entonces cómo es posible que Node sea asíncrono si V8 que es en lo que se basa en síncrono?

Ya responderemos a esa pregunta, pero primero tenemos que aclarar que es asíncrono y síncrono en cuanto a código se refiere.

**Programación síncrona**

Tradicionalmente en programación, la mayoría de procesos de entrada/salida ocurren de manera síncrona; como leer un archivo de texto, hacer una petición a una base de datos, etcétera. La ejecución del código síncrono se hace línea por línea, una tras otra. Por ejemplo, si nos basamos en Java y en cómo se leería un archivo usándolo, se necesitaría algo como esto:

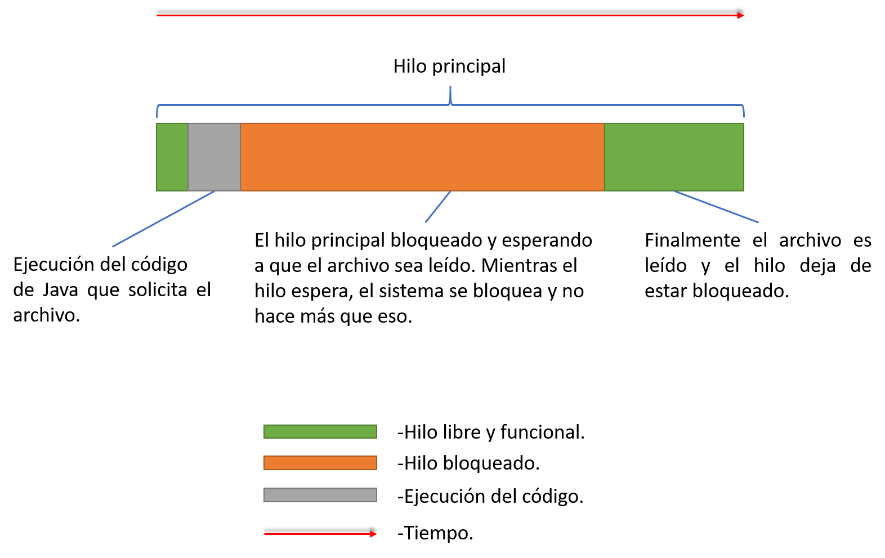
try( **FileInputStream** inputStream = new FileInputStream("archivo.txt") ) {

**Session** IOUtils;

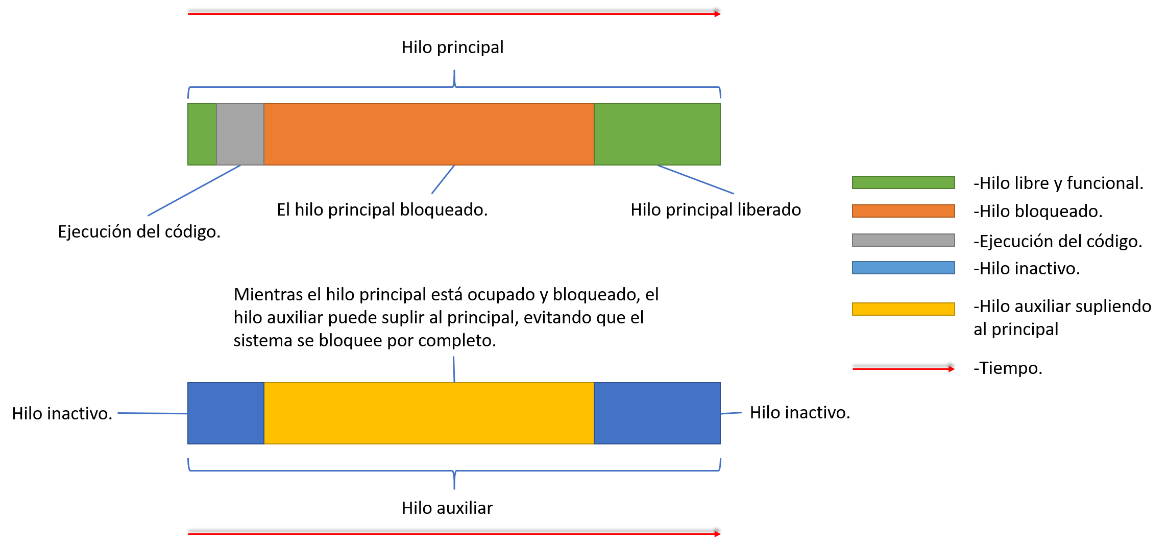
**String** fileContent = IOUtils.toString( inputStream );

}

En el caso de que ejecutáramos el anterior código en Java, lo que pasaría internamente sería que el hilo principal permanecería bloqueado, esperando a que el sistema lea el archivo desde el disco duro del ordenador, y lo pase a memoria principal; como lo muestra la siguiente ilustración:



Una de las soluciones a este problema sería la introducción de la programación multithreading (programación con múltiples hilos), como lo muestra la siguiente imagen:



La solución multithreading se puede considerar programación asíncrona, ya que hacemos operaciones asíncronamente gracias a la ayuda de múltiples hilos. Esta implementación es utilizada por muchos lenguajes de programación, pero tiene la gran desventaja de tener que manejar manualmente los hilos y tenerlos sincronizados. Tal vez no sea un gran problema cuando tengamos sólo dos hilos o diez hilos o en proyectos no empresariales (proyectos que no necesitan gran eficiencia), pero… ¿qué pasa cuando tengamos que tener sincronizados miles de hilos? La complejidad del manejo de tantos hilos es una tarea realmente exigente.

Por eso mismo Node.js implementa un enfoque diferente de un solo hilo… Programación asíncrona en Node.js.

## **Programación Asíncrona en Node.js**

El enfoque asíncrono de I/O que implementa Node.js es una forma de procesamiento de “input/output (entrada/salida)” que permite que múltiples procesos trabajen en un solo hilo principal. Mientras un proceso de I/O está esperando la respuesta del sistema operativo (por ejemplo, la petición de un archivo que está en el disco duro), dicho proceso se queda a la espera, pero no bloquea el hilo principal, permitiéndole a otros procesos seguir haciendo uso del hilo principal. La misma implementación de la lectura de un archivo de manera asíncrona en Node.js sería la siguiente:

*//Módulo del núcleo de Node que permite el manejo de archivos.*

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

*//Método del módulo 'fs'*

fs.readFile('archivo.txt', (err, contenido) **=>** {

*//Comprobamos si hay algún error.*

if (err) throw err;

*//Imprimimos el contenido del archivo.*

console.log("Contenido del archivo leído en línea 10.", contenido.toString("utf8"));

});

*//Imprimimos que el código se ha terminado.*

console.log("Final del código - Línea 14.");

Bien, ahora explicaremos cómo funciona el código anterior.

Primero importamos el módulo ***fs,*** el cual ya viene integrado en el núcleo de Node.js, lo que significa que no tenemos que instalarlo con NPM.

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

Después llamamos a la función ***readFile*** la acepta como parámetro el *path* de nuestro archivo(es decir, la ubicación) y como segundo parámetro acepta una ***callback*** que será ejecutada una vez que el archivo sea leído. Esta ***callback*** tiene como parámetros *err* y *contenido.* Si ***readFile*** tuviese un error mientras lee el archivo, pasaría un error a nuestra ***callback*** y el parámetro *contenido* tendría valor de *null o indefinido.* Sin embargo, si el archivo es leído correctamente, al final la función de ***readFile*** pasaría el valor de *null* en nuestro parámetro de *err*, y en *contenido* estaría el contenido de nuestro archivo en formato de buffer.

fs.readFile('archivo.txt', (err, contenido) **=>** {

. . .

}

Después, dentro de la ***callback*** verificamos si el *err* tiene algún valor, si el valor de *err* es *null,* significaría que ***readFile*** pudo leer satisfactoriamente el archivo y el código “throw err;” no sería ejecutado. Si la función ***readFile*** no pudiera leer correctamente el archivo, entonces *err* contendría el error y por lo tanto el ***if*** se evaluaría como verdadero, lanzando un error.

if (err) throw err;

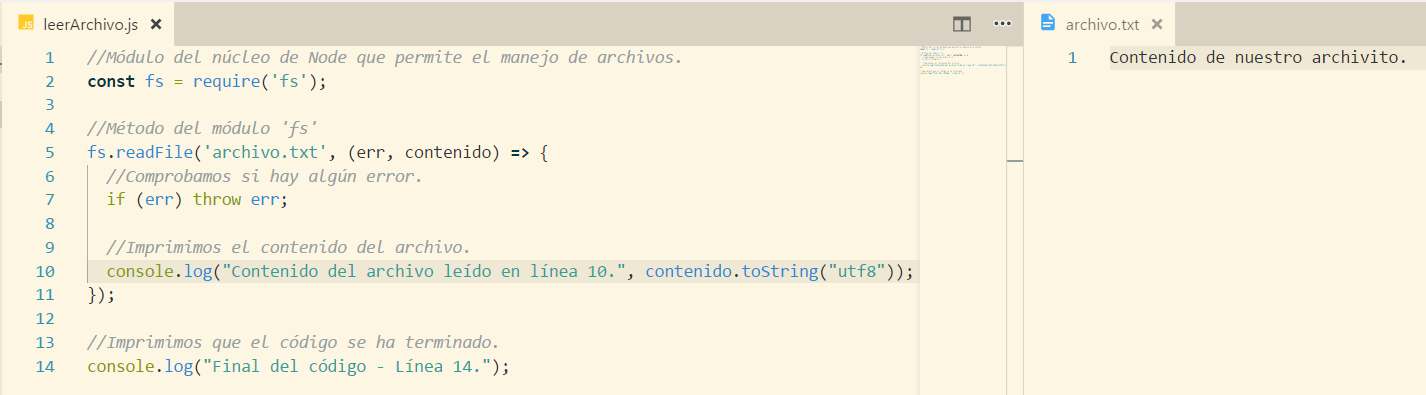
En caso de que no hubiese ningún error, el siguiente *console.log* imprimiría el texto que contiene y el contenido de nuestro variable *contenido*. Ya que la variable *contenido* es de tipo *buffer*, tenemos que convertirla a formato de *string* basándonos en la codificación *utf8.*

console.log("Contenido del archivo leído en línea 10.", contenido.toString("utf8"));

Finalmente, fuera de nuestra ***callback***, imprimimos un último *console.log* que indica que es el final del código.

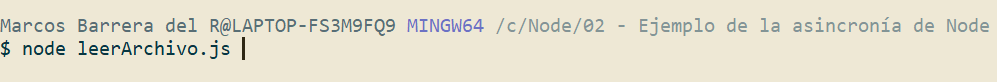
console.log("Final del código - Línea 14.");

Bien, en nuestro editor de texto, hemos creado un archivo de JavaScript que contiene el código al igual que creamos un archivo de texto llamando “archivo.txt” en el mismo directorio. Este archivo será leído por Node.js y una vez que sea leído, se ejecutará un *console.log* que imprimirá su contenido.

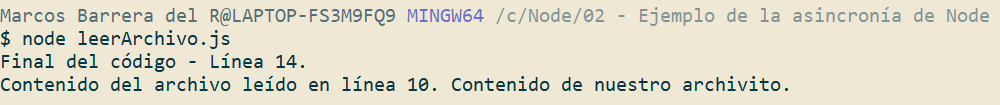


Como podemos ver, tenemos dos ***console.log.*** En programación síncrona como la de Java normalmente pensaríamos que el ***console.log*** de la línea 10 se mostraría primero y después el de la línea 14. Buen, probémoslo.

En nuestra terminal/consola de preferencia ejecutaremos el código de la siguiente manera:



Y daremos Enter para ver el resultado.



Como podemos ver, la línea 14 se ejecutó primero que la línea 10, ¿cómo es posible esto? Bueno, esta es la naturaleza asíncrona de Node.js. Ya que en nuestro código hicimos un proceso de I/O (Entrada/Salida), lo que hizo Node fue solicitar el archivo al sistema operativo, pero en lugar de esperar y bloquear el hilo principal hasta el que sistema operativo respondiese, dejó en espera ese proceso y continuó con el siguiente hasta que estuviera listo el archivo. Por ello, el código que se encontraba dentro de la ***callback*** se ejecutó después que el que estaba después de la ***callback***. A este patrón se le llama Non-Blocking. Una analogía bastante acertada sería la siguiente:

Imaginemos que hay un chef en un restaurante, un mesero y llega un cliente. El cliente realiza su pedido, el mesero anota el pedido y lo lleva con el chef, pero este mesero no es muy perspicaz que digamos y cada vez que lleva una orden al chef, se queda esperando junto a él hasta que dicho chef termina de cocinar lo que pidió el cliente. Ahora imaginemos que llegan otros diez clientes que esperan ser atendidos, pero como nuestro mesero sigue esperando a que el chef termine de cocinar, estos nuevos diez clientes tendrán que esperar hasta que el chef finalice. Supongamos ahora que el chef terminó y el mesero le lleva la comida al primer cliente y continúa con el primero de los diez clientes que llegaron. El dichoso mesero volvería a llevar el nuevo pedido al chef y se volvería a esperar con el chef hasta que éste termine de cocinar el pedido. Esto significa que los nueve clientes faltantes tendrían que esperar muchísimo tiempo para ser atendidos… una terrible atención al cliente.

Esta analogía representaría lo que es la programación síncrona con un solo hilo. Los clientes son navegadores web haciendo peticiones HTTP a una página web, el mesero es un hilo que se bloquea cada vez que recibe una petición (como un único hilo de Java) y el chef sería el sistema operativo tratando de entregar el contenido de un archivo que se encuentre en el disco duro o una petición a una vez de datos que puede tardar tiempo en ser ejecutada.

Ahora bien, ¿cómo podríamos solucionar este problema en esta analogía? Contratando más meseros (es decir, creando más hilos). De esta manera cada vez que llegue un nuevo cliente, uno de los meseros lo atendería, y si aun así llegan más clientes, podemos hacer uso de los nuevos meseros que contratamos. Perfecto. Pero ahora tenemos otros dos problemas, el primero es que tenemos que pagarle su sueldo a cada uno de los meseros nuevos que contratemos (es decir, cada uno de estos nuevos hilos requerirán de memoria RAM y procesamiento en el CPU), y el segundo es que tenemos que mantener en sincronía a cada uno de nuestros meseros para que puedan trabajar en equipo (es decir, tendríamos que sincronizar cada uno de nuestros hilos para que ejecuten los procesos en el orden que queremos).

Ahora bien, ¿Qué pasaría si en vez de tener meseros ineficientes (múltiples hilos), tuviésemos a un solo mesero inteligente y eficiente (el hilo único que implementa Node)? Pues en este caso solucionaríamos los anteriores dos problemas, ya que sólo tendríamos que pagarle a un solo mesero, lo que sería más económico; y ya no tendríamos que sincronizar a todo el montón de meseros, ya que, en este caso, solo tendríamos a un mesero.

Es decir, cada vez que un cliente llegue al restaurante, nuestro único mesero inteligente tomaría el pedido y lo llevaría con el chef, pero esta vez, nuestro mesero no se quedaría esperando a que el chef termina, sino que regresaría con los clientes para ver si hay otro cliente por atender. De esta manera mejoramos drásticamente la eficiencia de nuestro restaurante.

Nuevamente, traduciendo esto al ambiente de programación, los clientes son navegadores web haciéndole peticiones a nuestro servidor web (nuestro restaurante), Node.js y su hilo único (nuestro mesero) atiende cada una de esas peticiones y se las otorga al Sistema Operativo o a la Base de Datos (el chef); mientras estas peticiones esperan, Node sigue a la escucha de nuevas peticiones. Una vez que el sistema operativo o la base de datos responden (el chef termina el pedido), Node toma la información resultante y se la entrega a cada uno de los clientes. Sencillo, ¿no?

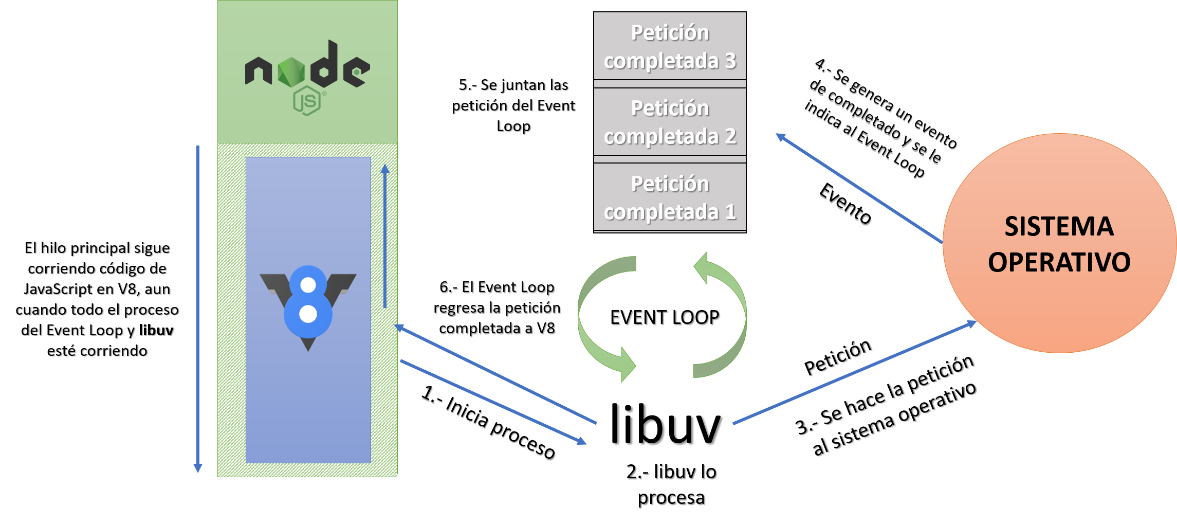
Ahora que sabemos cómo funciona Node.js asíncronamente y cómo funciona V8, falta responde a la pregunta que dejamos en el aire anteriormente… ¿Cómo es posible que Node sea asíncrono si V8 que es en lo que se basa en síncrono?

La respuesta a esta pregunta se llama **libuv.**

**Libuv**

Simplificando, libuv es una librería multiplataforma con enfoque en procesos de I/O (Input/Ouput o Entrada/Salida) asíncronos. Esta librería fue desarrollada para ser usada en Node.js, pero puede ser utilizada en otros entornos.

La manera en la que libuv encaja en Node.js es la siguiente:



Aunque a lo largo de la guía nos hemos referido a Node.js como código de JavaScript meramente, técnicamente una porción de Node está escrita en C++, de esta manera Node se puede comunicar más fácilmente con el motor V8. Bien, ahora explicaremos el proceso de toda la asincronía de Node.js y cómo funciona ***libuv*** en ella.

1.- El proceso inicia cuando Node hace una petición a V8 y V8 realiza la petición a **libuv**.

2.- **libuv** hace de intermediario entre V8 y el Sistema Operativo (SO).

3.- El SO es el que tiene la capacidad de leer archivos, hacer peticiones HTTP por medio de internet, etc. Por lo cual se delega el trabajo pesado a él.

4.- Una vez que el SO termina de procesar la petición, genera un evento y lo pone en una cola, como una petición completada.

5.- Las peticiones completadas se van acumulando para que el Event Loop las procese posteriormente.

6.- El Event Loop procesa cada una de las peticiones completadas y las regresa a V8. Finalmente, V8 regresa el resultado a Node y termina el proceso.

Ahora que ya vimos cómo funciona Node.js internamente, podemos proceder finalmente con la creación de un servidor web de tipo API y algunos de los frameworks de Node.js.

**Fuentes consultadas:**

<https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/>

[Curso “Learning and Understanding NodeJS” de Anthony Alicea](https://www.udemy.com/understand-nodejs/learn/v4/overview)

<https://nodejs.org/es/about/>

<https://blog.risingstack.com/node-hero-async-programming-in-node-js/>

**Web Frameworks**

Antes de profundizar en el desarrollo de una API (concepto explicaremos dentro de poco), tenemos que tener claro qué es un *Web Framework,* ya que usaremos uno llamada Express, por lo cual, un poco de trasfondo podrá ayudar a comprender mejor el panorama.

Un *Web Framework* es un conjunto de componentes diseñado para simplificar el proceso de desarrollo web. Tiene determinadas herramientas integradas que sirven cómo fundamento para un proyecto web. Un *Web Framework* permite que nosotros como desarrolladores podamos enfocarnos en los detalles más importantes y específicos de nuestro proyecto, en lugar de construir toda la estructura básica inicial de un proyecto.

Por ejemplo, si quisiéramos crear una aplicación web medianamente compleja, seguramente necesitaríamos crear un form para loguear a nuestros usuarios, necesitaríamos manejar bases de datos, integrar con redes sociales, construir una seguridad robusta y unas cuantas cosas más. Entonces, en lugar de desarrollar todos estos componentes nosotros mismos, podríamos utilizar un *Web Framework* que ya tiene integrado cada uno de estos componentes o que facilidad la construcción de los mismos enormemente.

**Verbos/Métodos HTTP**

Antes de mencionar qué *Web Framework* de Node.js, explicaremos un poco qué son los verbos/métodos HTTP y daremos unos ejemplos. Todo esto con el fin de clarificar temas que usaremos frecuentemente en posteriores secciones.

HTTP define un conjunto de métodos a sus peticiones para indicar la acción deseada a realizar en una aplicación web de lado de servidor. Aunque algunos de ellor pueden ser sustantivos, estos métodos normalmente son llamados como verbos HTTP (HTTP verbs). Cada uno de ellos implemente una semántica diferente. Sin embargo, estos verbos HTTP comparte algunas características entre sí.

**GET:** El método GET solicita una representación de un recurso (registro) especificado. Un método GET debería ser utilizado sólo para solicitar información, no para proporcionarla.

Por ejemplo, cada vez que ingresamos a [www.google.com](http://www.google.com), nuestro navegador web realiza una petición (*request*) GET a los servidores de Google, solicitando el contenido de la página principal del buscador. Un determinado servidor de Google le responde (*response*) a nuestro navegador web con la página HTML del buscador, nuestro navegador renderiza dicha página y nos muestra su contenido.

**POST:** El método POST es usado para enviar información a un servidor dado, normalmente para registrar información en dicho servidor.

Por ejemplo, cada vez que nos *logueamos* en una página web, nuestro navegador web envía una petición (*request*) POST a los servidores de dicha página web con nuestro usuario y contraseña. Un determinado servidor recibe la petición POST y lee su contenido, comprueba que nuestras credenciales sean correctas al compararlas con las que tiene en su base de datos. Si las credenciales son correctas, nos responde (*response*) un *status code* (normalmente de valor “200”) y nos permite ingresar a nuestra cuenta, de lo contrario, nos responde con un *status code* de tipo error y nos deniega el acceso a la cuenta.

**PUT:** El método PUT reemplaza un registro dado con uno nuevo que se envíe en el cuerpo de la petición (*request*).

Por ejemplo, si un empleado de Wallmart quisiera editar la información de un producto mediante su navegador web, localizaría el producto en el sistema y el sistema internamente capturaría el ID del producto, el empleado vería la información actual del producto y editaría la información que desease. Después enviaría la nueva información y el ID del producto al servidor mediante una petición (*request*) PUT al servidor, el servidor buscaría el registro actual basándose en el ID que la petición le envió, lo localizaría y reemplazaría toda la información actual con la información enviada en la petición. Técnicamente sería el mismo registro, ya que el ID permanecería intacto, pero dicho registro habría sido modificado. Opcionalmente (y de manera recomendada), el servidor le respondería (*response*) al navegador web del empleado con el *status code* para indicarle si la petición PUT fue atendida correctamente o si tuvo algún error.

**DELETE:** El método DELETE borra el recurso (registro) especificado.

Siguiendo el ejemplo del método PUT. Si el mismo empleado de Wallmart quisiera borrar un producto del sistema, ingresaría al sistema con su navegador web, buscaría el producto y una vez localizado y seleccionado, el sistema internamente guardaría el ID del producto, si el empleado confirma la eliminación del producto, el navegador web realizaría una petición (*request*) DELETE junto con el ID del producto a un servidor. El servidor recibiría dicha petición e interpretaría que es una de tipo DELETE, lo cual le indicaría al servidor que borre el registro que concuerde con el ID proporcionado. Una vez borrado el registro de la base de datos, el servidor respondería (*response*) con el *status code* de la petición para indicarle al navegador web si la petición fue satisfecha correctamente o si hubo un error de por medio.

Los anteriores verbos/métodos HTTP son los que utilizaremos a lo largo de la guía, pero eso no significa que no existan más. Algunos otros verbos HTTP que existe son CONNECT, OPTIONS, TRACE, PATCH, HEAD, etc.

**Express**

Ahora que tenemos un poco más claro que es un *Web Framework* y en qué consisten los verbos HTTP, introduciremos Express, el framework de Node.js que usaremos en esta guía. De manera resumida, Express es un *Web Framework* muy popular escrito en JavaScript y ejecutado dentro del entorno de ejecución de Node.js.

Usualmente, una aplicación web moderna espera recibir peticiones HTTP de un navegador web (u otro tipo de cliente). Cuando una petición es recibida, la aplicación tiene que deducir que acción se necesita ejecutar basándose en el patrón que presente la URL y la posible información que pueda contener la petición, ya sea de tipo POST o de tipo GET. Dependiendo de qué se requiera, la aplicación quizá lea o grabe información en una base de datos, suba un archivo, envíe un correo electrónico, entre otros; para poder satisfacer la petición recibida. Dicha aplicación al final del proceso retornará una respuesta hacia el cliente (un navegador web, un dispositivo móvil…), y esta respuesta puede contener una página web HTML o puede contener información en formato JSON.

Express provee métodos para especificar cuál función será llamada, basándose en el verbo HTTP (GET, POST, PUT, DELETE, etc.) y en el patrón de la URL (*Route o* ruta) de una determinada petición HTTP.

**Qué es una API**

Antes de seguir con la guía, nos gustaría explicar que es una API, ya que es un término que utilizaremos en las posteriores secciones de la guía.

En computación, una API es una lista de comandos con un determinado formato que un programa puede enviar a otro y viceversa. Una API es usada para que programas independientes puedan enviarse información o ejecutar funciones los unos con los otros.

Por ejemplo, el programa “Bloc de Notas” o “Notepad” (integrado en S.O. Windows) puede comunicarse con Windows diciéndole que imprima un documento. Esto se puede hacer de muchas formas, pero la forma más básica es llamar a la función/comando:

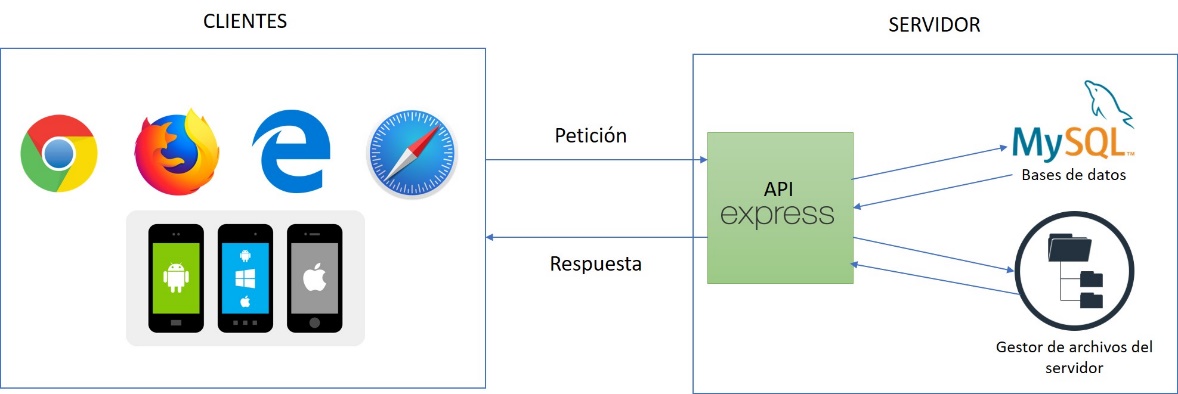
***print(document);***

Esta característica es realmente útil en multitud de manera. Por ejemplo, esta característica le permite al programa de *Notepad* ser capaz de imprimir documentos sin tener que comunicarse directamente con la impresora. Todo el trabajo de comunicación entre un programa independiente (como *Notepad*) y la impresora, al igual que toda la gestión de recursos (memoria RAM, procesador, disco duro) que se requieren para poder imprimir es manejado por Windows. Todo lo que un programa independiente como *Notepad* tiene que hacer es ejecutar el comando ***print()***.

Las APIs son provistas por cualquier programa que permita interacción con otros programas. Las personas que escriben programas, y que desean utilizar funcionalidades que ya existen en otros programas (a través de APIs de dichos programas), simplemente pueden echar un vistazo a la documentación de la APIs para encontrar qué comandos están disponibles.

Ahora bien, en el contexto de la Web, las APIs generalmente nos permiten enviar comandos a programas/aplicaciones que están corriendo en servidores web, y dichas APIs nos sirven como intermediario entre nuestro navegador y los diferentes recursos que nos pueda ofrecer un servidor web; esto recursos podrían ser archivos que tiene en disco duro el servidor, registros de la base de datos del servidor en cuestión, etc.

En *Express* nosotros podemos programar una API con la que proporcionaremos funciones/comandos mediante peticiones HTTP y sus posibles verbos HTTP. Es decir, al programar una aplicación en *Express,* creamos una capa intermediaria entre el cliente (nuestro navegador web, por ejemplo) y los distintos componentes de nuestro servidor web (el sistema de archivos, la base de datos, etc.), como en la siguiente imagen.



En resumen, podemos ver cómo los clientes hacen peticiones (*requests*) al servidor, y una *API de Express* es la intermediaria entre los clientes y la base de datos o el gestor de archivos.

Este tipo de arquitectura, si está bien programada, aumenta la seguridad de nuestro servidor, ya que los clientes sólo tienen acceso a las acciones que nosotros hayamos establecido en nuestra API.

Si un cliente quisiera borrar una tabla de nuestra base de datos, no podría, ya que el único acceso que tiene hacia la base de datos es mediante nuestra API de *Express*, sobre la cual sólo nosotros tenemos control.

**Fuentes consultadas:**

[**https://www.quora.com/What-is-an-API-4**](https://www.quora.com/What-is-an-API-4)

[**https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods)

[**https://djangostars.com/blog/what-is-a-web-framework/**](https://djangostars.com/blog/what-is-a-web-framework/)

[**https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\_Nodejs/Introduction**](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction)