NODE.JS y MONGODB
CLASE 1 : CONCEPTOS BÁSICOS

NODEJS

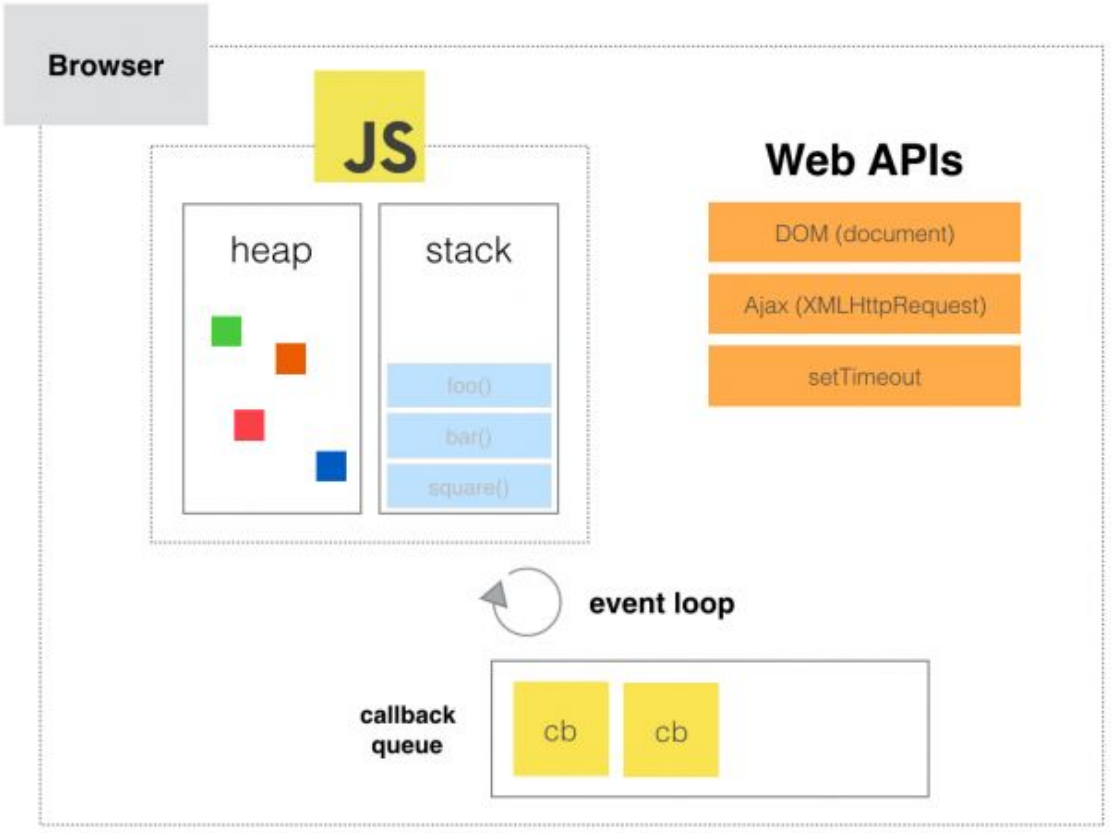
Node.js es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Node.js usa un modelo de operaciones E/S sin bloqueo y orientado a eventos, que lo hace liviano y eficiente. El ecosistema de paquetes de Node.js, npm, es el ecosistema más grande de librerías de código abierto en el mundo.

	NodeJS	Ruby	Python	PHP
Lenguaje	Javascript	Ruby	Python	PHP
Motor	V8,Chakra,TraceMonkey	YARV	cPython	Apache
Entorno	Modulos de Núcleo	Librerías Externas	Librerías Externas	Librerías Externas
Frameworks	Meteor,Express	Rails	Django	Laravel

EVENT LOOP

Un modelo orientado a eventos es aquel que maneja concurrencia enviando un mensaje cuando un evento se acciona. El mismo sale “afuera”(Web APIs) para proceso interno lo cual le da lugar a recibir el próximo mensaje.

Cuando la solicitud se ha completado, se devuelve a la cola de procesamiento,en donde al alcanzar el comienzo, el resultado se devuelve al usuario.



PATRÓN REACTOR

El patrón de diseño reactor es un patrón de programación concurrente para manejar los pedidos de servicio entregados de forma concurrente a un manejador de servicio desde una o más entradas. El manejador de servicio demultiplexa los pedidos entrantes y los entrega de forma sincrónica a los manejadores de pedidos asociados.

Reactor Pattern es una idea de las operaciones de I/O sin bloqueo en Node.js. Este patrón proporciona un controlador (en el caso de Node.js, una función callback) que está asociado con cada operación de I/O. Cuando se genera una solicitud de I/O, se envía a un demultiplexor.

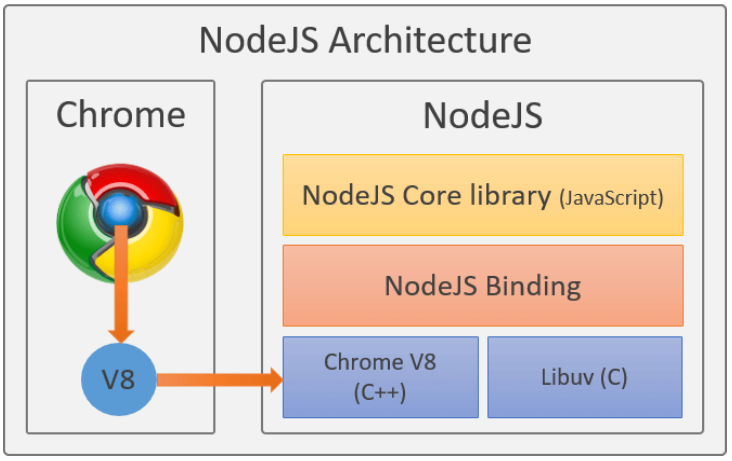
Este demultiplexor es una interfaz de notificación que se utiliza para controlar la concurrencia en el modo de I/O sin bloqueo y recopila cada solicitud en forma de un evento y pone en cola cada evento en una cola. Por lo tanto, el demultiplexor proporciona la cola de eventos, que a menudo escuchamos. Cuando el

demultiplexor recoge una solicitud, devuelve el control al sistema y no bloquea la I/O. Al mismo tiempo, hay un bucle de eventos que itera sobre los elementos en la cola de eventos. Cada evento tiene un callback asociado, y ese callback se invoca cuando itera el bucle de evento.

El callback además tiene, en su mayoría, otros callback asociados que representan algunas operaciones asíncronas. Estas operaciones se insertan en la cola de eventos por el demultiplexor y están listas para ser procesadas una vez que el bucle de evento las itera. Es por eso que las llamadas a otras operaciones deben ser asincrónicas.

Cuando se procesan todos los elementos en la cola de eventos y no quedan operaciones pendientes, Node.js finaliza la aplicación automáticamente.

ARQUITECTURA DE NODE.JS

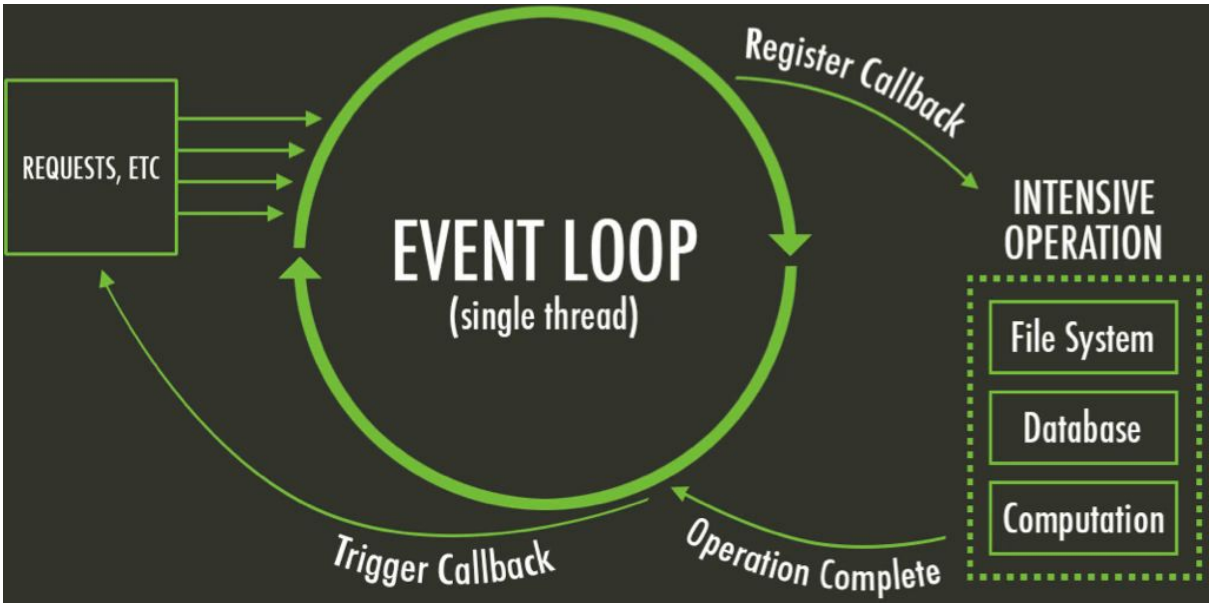


La estructura interna de Node.js está compuesta por los siguientes elementos y en el siguiente orden de ejecución :

- **Motor V8(engine)**
- **Compilador de código C++ y C** para traducción de archivos .js a lenguaje máquina
- **Núcleos** de la misma librería Node.js que no se importan como librerías externas sino que ya son parte del mismo entorno por defecto

El hecho de que el código de Node.js esté escrito en C++ y C es lo que hace que el entorno sea de alta performance en cuanto a resolver respuestas.

Por otro lado, Node.js es una tecnología asincrónica (async), lo que significa que los flujos de ejecución no son procedimentales en todas las operaciones.
Maneja un solo "thread" (proceso), el cual está en un loop constante verificando si se han disparados eventos. Cada evento tiene asociados N cantidad de callbacks.
Es un modelo tradicional de emitters y listeners.



Como node.js es una tecnología muy usada para construir aplicaciones de red, nos basaremos en "Request" (visitas de usuario) a las mismas.
En el gráfico podemos ver el evento loop, el cual está constantemente monitoreando si un evento ha sido disparado.
Cada vez que un evento es lanzado los callbacks asociados a los mismos son ejecutados.
Los procesos asincrónicos son operaciones que se ejecutan "en paralelo" al flujo de ejecución de al apliaciones. Cuando un proceso asincrónico termina el mismo dispara un evento que es asociado al callback.
En Node.js las operaciones de acceso al sistema de archivo, bases de datos o trabajo en red son normalmente procesos asincrónicos, es por ello que las funciones (métodos) para operar en forma asincrónica llevan como argumento una función (callback).

.

ECMASCRIPT 6

ECMAScript es una especificación de lenguaje de programación publicada por ECMA International. El desarrollo empezó en 1996 y estuvo basado en el popular

lenguaje JavaScript propuesto como estándar por Netscape Communications Corporation. Actualmente está aceptado como el estándar ISO 16262.
ECMAScript define un lenguaje de tipos dinámicos ligeramente inspirado en Java y otros lenguajes del estilo de C. Soporta algunas características de la programación orientada a objetos mediante objetos basados en prototipos y pseudoclases.

La mayoría de navegadores de Internet incluyen una implementación del estándar ECMAScript, al igual que un acceso al Document Object Model para manipular páginas web. JavaScript está implementado en la mayoría de navegadores, Internet Explorer de Microsoft usa JScript. El navegador Opera tenía su propio intérprete de ECMAScript con extensiones para soportar algunas características de JavaScript y JScript, actualmente Opera está basado en Chromium (y utiliza su intérprete). Cada navegador tiene extensiones propias del estándar ECMAScript, pero cualquier código que se adecúe al estándar debería funcionar en todos ellos.

ActionScript, para Adobe Flash, también está basado en el estándar ECMAScript, con mejoras que permiten mover, crear y analizar dinámicamente objetos, mientras la película está en ejecución.
Desde el lanzamiento en junio de 1997 del estándar ECMAScript 1, han existido las versiones 2, 3 y 5, que es la más usada actualmente (la 4 se abandonó 2). En junio de 2015 se cerró y publicó la versión ECMAScript 6

Abordemos entonces primero la sintaxis de esta última definición ya que la mayoría de las nuevas especificaciones son realmente usadas.

CONSTRUCTORES LET Y CONST

let

La instrucción let declara una variable de alcance local con ámbito de bloque(block scope), la cual, opcionalmente, puede ser inicializada con algún valor.

Sintaxis :

```
let var1 [= valor1] [, var2 [= valor2]] [, ..., varN [= valorN]];
```

let permite declarar variables limitando su alcance (scope) al bloque, declaración, o expresión donde se está usando. Lo anterior diferencia let de la palabra

reservada var , la cual define una variable global o local en una función sin importar el ámbito del bloque.
Variables declaradas por **let** tienen por alcance el bloque en el que se han definido, así mismo, como en cualquier bloque interno. De esta manera, **let** trabaja muy parecido a var. La más notable diferencia es que el alcance de una variable var es la función contenedora :

```
function varTest() {
  var x = 31;
  if (true) {
    var x = 71;  // ¡misma variable!
    console.log(x);  // 71
  }
  console.log(x);  // 71
}

function letTest() {
  let x = 31;
  if (true) {
    let x = 71;  // variable diferente
    console.log(x);  // 71
  }
  console.log(x);  // 31
}
// llamamos a las funciones
varTest();
letTest();
```

En el nivel superior de un programa y funciones, let , a diferencia de var, no crea una propiedad en el objeto global, por ejemplo:

```
var x = 'global';
let y = 'global';
console.log(this.x); // "global"
```

```
console.log(this.y); // undefined
```

La redeclaración de la misma variable bajo un mismo ámbito léxico terminaría en un error de tipo SyntaxError. Esto también es extensible si usamos var dentro del ámbito léxico. Esto nos salvaguarda de redeclarar una variable accidentalmente y que no era posible solo con var.

```
if (x) {
  let foo;
  let foo; // Terminamos con un SyntaxError.
}
if (x) {
  let foo;
  var foo; // Terminamos con un SyntaxError.
}
```

En ECMAScript 2015, let no eleva la variable a la parte superior del bloque. Si se hace una referencia a la variable declarada con let (let foo) antes de su declaración, terminaríamos con un error de tipo ReferenceError (al contrario de la variable declarada con var, que tendrá el valor undefined), esto porque la variables vive en una "zona muerta temporal" desde el inicio del bloque hasta que la declaración ha sido procesada.

```
function do_something() {
  console.log(bar); // undefined
  console.log(foo); // ReferenceError: foo no está
definido
  var bar = 1;
  let foo = 2;
}
```

Es posible encontrar errores en bloques de control switch debido a que solamente existe un block subyacente.

```
switch (x) {
  case 0:
    let foo;
  break;

  case 1:
    let foo; // Terminamos con un error de tipo
SyntaxError.
             // esto debido a la redeclaracion
  break;
}
```

const

Las variables constantes presentan un ámbito de bloque(block scope) tal y como lo hacen las variables definidas usando la instrucción let, con la particularidad de que el valor de una constante no puede cambiarse a través de la reasignación. Las constantes no se pueden redeclarar.
La redeclaración de la misma variable bajo un mismo ámbito léxico terminaría en un error de tipo SyntaxError. Esto también es extensible si usamos var dentro del ámbito léxico. Esto nos salvaguarda de redeclarar una variable accidentalmente y que no era posible solo con var.
Sintaxis :

```
const varname1 = value1 [, varname2 = value2 [, varname3 = value3 [, ...
[, varnameN = valueN]]]];
```

Esta declaración crea una constante cuyo alcance puede ser global o local para el bloque en el que se declara. Es necesario inicializar la constante, es decir, se debe especificar su valor en la misma sentencia en la que se declara, lo que tiene sentido, dado que no se puede cambiar posteriormente.

La declaración de una constante crea una referencia de sólo lectura. No significa que el valor que tiene sea inmutable, sino que el identificador de variable no puede ser reasignado, por lo tanto, en el caso de que la asignación a la constante sea un objeto, el objeto si que puede ser alterado.
Una constante no puede compartir su nombre con una función o variable en el mismo ámbito.

Todas las consideraciones acerca de la " zona muerta temporal " se aplican tanto a let y const.

FUNCIONES FLECHA

La expresión de función flecha tiene una sintaxis más corta que una expresión de función convencional y no vincula sus propios this, arguments, super, o new.target. Las funciones flecha siempre son anónimas. Estas funciones son funciones no relacionadas con métodos y no pueden ser usadas como constructores.

Sintaxis :

```
(param1, param2, ..., paramN) => { sentencias }
(param1, param2, ..., paramN) => expresion
// Equivalente a: () => { return expresion; }

// Los paréntesis son opcionales cuando sólo dispone de un argumento:
singleParam => { statements }
(singleParam) => { sentencias } singleParam => { sentencias }

// Una función sin argumentos requiere paréntesis:
() => { sentencias }
```

La inclusión de las funciones flecha se vieron influenciadas por dos factores: sintaxis reducida y this no vinculable.

Antes a las funciones flecha, cada nueva función definía su propio valor de this (un nuevo objeto en el caso de un constructor, undefined en llamadas a funciones en modo estricto, el objeto contextual si la función se llama como un "método de un objeto", etc.). Lo cual resultaba molesto cuando se intentaba aplicar programación orientada a objetos.
Dado que las funciones flecha no tienen su propio this, los métodos call() y apply()sólo pueden pasar parámetros. thisArg es ignorado.

Cuerpo de una función flecha

Las funciones flecha pueden tener ya sea un "cuerpo conciso" o el usual "cuerpo de bloque".
En un cuerpo conciso, solo una expresión es especificada, la cual convierte el valor de retorno explícito. en un bloque cuerpo, tu tienes que usar una declaración de retorno explícita.

```
var func = x => x * x;
// sintaxis de cuerpo conciso, el "return" está implícito

var func = (x, y) => { return x + y; };
// con cuerpo de bloque, se necesita "return" explícito
```

Tenga en cuenta que retornar objetos literales usando la sintaxis simplificada params => objeto:literal} no funciona como se esperaría:

```
var func = () => {  foo: 1  };
// Al llamar func() retorna undefined!

var func = () => {  foo: function() {}  };
// Error de sintaxis: SyntaxError: function statement
requires a name
```

Eso es debido a que el código dentro de las llaves ({}) es leído como una secuencia de sentencias (p.ej foo es tratado como una etiqueta o label y no como una propiedad del objeto literal).
En este caso, recuerde encerrar el objeto literal entre paréntesis:

```
var func = () => ({ foo: 1 });
```

CONDICIONAL TERNARIO

El operador condicional (ternario) es el único operador en JavaScript que tiene tres operandos. Este operador se usa con frecuencia como atajo para la instrucción if.

Sintaxis :

```
condición ? expr1 : expr2
```

Si la condición es true, el operador retorna el valor de la expr1; de lo contrario, devuelve el valor de expr2. También es posible realizar evaluaciones ternarias múltiples (Nota: El operador condicional es asociativo):

```
var firstCheck = false,
    secondCheck = false,
    access = firstCheck ? "Acceso Denegado" : secondCheck
? "Acceso Denegado" : "Acceso Permitido";

console.log( access ); // muestra "Acceso Permitido"
```

OPERADOR DE PROPAGACION(SPREAD)

El operador de propagación spread operator permite que una expresión sea expandida en situaciones donde se esperan múltiples argumentos (llamadas a funciones) o múltiples elementos (arrays literales).
Sintaxis:

- Llamadas a funciones:

```
f(...iterableObj);
```

- Arrays literales:

```
[...iterableObj, 4, 5, 6]
```

- Desestructuración destructuring:

```
[a, b, ...iterableObj] = [1, 2, 3, 4, 5];
```

OPERADOR REST

El operador Rest es exactamente igual a la sintaxis del operador de propagación, y se utiliza para desestructurar arrays y objetos. En cierto modo, Rest es lo contrario de spread. Spread 'expande' un array en sus elementos, y Rest recoge múltiples elementos y los 'condensa' en uno solo.
La sintaxis de los parámetros rest nos permiten representar un número indefinido de argumentos como un arreglo.

Sintaxis :

```
function(a, b, ...theArgs) {
  // ...
}
```

Los parámetros rest pueden ser desestructurados, eso significa que sus datos pueden ser desempaquetados dentro de distintas variables. Ver Destructuring assignment.

```
function f(...[a, b, c]) {
  return a + b + c;
}

f(1)          // NaN (b y c son indefinidos)
f(1, 2, 3)    // 6
f(1, 2, 3, 4) // 6 (el cuarto parámetro no está
desestructurado)
```

En el siguiente ejemplo, se usa un parámetro rest para agrupar todos los argumentos después del primero. Luego cada uno es multiplicado por el primero y el arreglo es regresado:

```
function multiply(multiplier, ...theArgs) {
  return theArgs.map(function (element) {
```

```
    return multiplier * element;
  });
}

var arr = multiply(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

CLOSURES

Los closures suelen ser uno de los temas más difíciles de entender en JavaScript, pero la realidad es que son un concepto simple.
En JavaScript podemos tener N funciones dentro de otra función, y cada función "contenida" tiene acceso a las variables de las funciones de nivel superior (contenedora), a eso se le llama closure

Veamos el ejemplo:

```
var main = function(){
   var name = "John";

     function child1(){
       console.log("hi "+name);
     }

     function child2(){
         console.log("bye "+name);
     }

     function change(){
       name = "Rita";
          child1();
          child2();
     }
```

```
    child1();
    child2();
    change();
}
main();
```

CLASES

Las clases de javascript son introducidas en el ECMAScript 2015 y son una mejora sintáctica sobre la herencia basada en prototipos de JavaScript. La sintaxis de las clases no introduce un nuevo modelo de herencia orientada a objetos a JavaScript. Las clases de JavaScript proveen una sintaxis mucho más clara y simple para crear objetos y lidiar con la herencia.

Las clases son de hecho "funciones especiales", tal y como el caso de las expresiones de funciones y declaraciones de funciones, la sintaxis de la clase tiene dos componentes:

- expresiones de clases
- declaraciones de clases.

Una manera de definir una clase es mediante una declaración de clase. Para la declaración de una clase, es necesario el uso de la palabra reservada class y un nombre para la clase ("Poligono" en este caso).

```
class Poligono {
  constructor(alto, ancho) {
    this.alto = alto;
    this.ancho = ancho;
  }
}
```

El método constructor es un método especial para crear e inicializar un objeto creado con una clase. Solo puede haber un método especial con el nombre "constructor" en una clase. Si esta contiene más de una ocurrencia del método constructor, se arrojará un Error SyntaxError.

Un constructor puede usar la palabra reservada super para llamar al constructor de una superclase

MÓDULOS EN NODE.JS

Los módulos en node.js son librerías escritas normalmente en javascript que nos permiten realizar operaciones de distintos tipos (conectarnos a una base de datos, crear un archivo comprimido, etc).

Para poder incluir un módulo usamos la función "require()" pasando como argumento el nombre del módulo a leer.

Los módulos en su gran mayoría retornan una función (clase).
Los callback (cb) normalmente reciben argumentos, el cual el primero siempre es el error (si no hay error se recibe un null)

PATRON MODULO

Para entender lo que un módulo puede brindarte, necesitas entender conceptualmente lo que hace esta porción de código:

```
(function () {
  // código
})();
```

Esto es una declaración de función, que luego se llama a sí mismo inmediatamente. Estos también se conocen como Immediately-Invoked-Function-Expressions o expresiones de función inmediatamente invocadas, en el que la función define el ámbito y crea la “privacidad”. JavaScript no tiene privacidad, pero la creación de nuevos ámbitos emula esto cuando envolvemos toda nuestra lógica de funcionamiento dentro de ellos. La idea entonces es devolver sólo las partes que necesitamos, dejando el otro código fuera del alcance global.

Después de crear un nuevo ámbito, necesitamos un espacio de nombres para nuestro código que permita acceder a cualquiera de los métodos que escribimos. Vamos a crear un espacio de nombres para nuestro módulo anónimo :

```
var Modulo = (function () {
  // código
})();

(function () {
  // código
window.Modulo - ...
})();
```

Entonces tenemos la variable Modulo declarada en el ámbito global, que significa que puede ser invocada por cualquiera que la necesite, o incluso pasarla a otro módulo. Vas a ver y oír mucho sobre métodos privados en JavaScript, pero en realidad nada es “privado” en Javascript, por tanto si necesitas estructurar tu código bajo este concepto, tenés que crearlo.

En Node.js este patrón viene por defecto, por lo que tenemos que entender que cada archivo que escribamos es un módulo por sí solo, es decir que sus variables son locales al módulo donde fueron declaradas y no pueden salir de ahí a menos que explícitamente lo declaremos. Veamos entonces las propiedades que puede tener cada módulo :

Globales

Método / Propiedad	Descripción
_dirname	El nombre del directorio del módulo actual *
_filename	El nombre del archivo del módulo actual *
clearImmediate(Object ref)	Cancela un Immediate Object creado con setImmediate
clearInterval(Object ref)	Cancela un Immediate Object creado con setInterval
clearTimeout(Object ref)	Cancela un Immediate Object creado con setTimeout

console	Objeto que contiene métodos para debugging similares a navegadores web
exports	Un objeto que transmite variables locales a través de la interfaz require()
global	El objeto global.
module	Una referencia al módulo actual
process	Objeto que provee información y controla los procesos de Node.js
require()	Requiere módulos *
setImmediate(Function callback[,...args])	Callback inmediato que se puede registrar a un evento luego de que este se disparó
setInterval(Function callback,delay[,...args])	Callback para disparar cada determinada cantidad de milisegundos
setTimeout(Function callback,delay[,...args])	Callback que se dispara después de como mínimo determinada cantidad de milisegundos

*Los métodos/propiedades marcadas con este símbolo implican que aparecen como globales para fines prácticos pero en realidad le pertenecen a cada módulo en particular.

HTTP

Vamos a comenzar con el módulo de HTTP. Las interfaces HTTP en Node.js están diseñadas para admitir muchas características del protocolo que tradicionalmente han sido difíciles de usar. En particular, mensajes grandes, posiblemente codificados en fragmentos. La interfaz tiene cuidado de nunca almacenar en el búfer todas las solicitudes o respuestas.

Las cabeceras de un objeto HTTP están representadas por un objeto como este:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
```

```
  'accept': '*/*' }
```

Este módulo contiene muchos métodos y propiedades pero de momento nos vamos a detener en el método capaz de crear un servidor.

```
http.createServer([Object options][, Function
requestListener])
```

- options : Un objeto de configuración que admite las propiedades IncomingMessage y ServerResponse para asignación de clases dinámicas
- requestListener : Una función callback que se ejecuta en cada solicitud HTTP hecha al servidor. Esta función es automáticamente agregada a la cola del evento 'request'
- Retorna : Una nueva instancia de http.Server

requestListener

Dado que el callback del método createServer es agregado a la cola del evento 'request' , el mismo va a contar con dos variables pre inicializadas 'request' y 'response' respectivamente en ese orden, las cuales pueden ser usadas para conocer más información sobre el status de la respuesta, headers , etc.

Método / Propiedad	Descripción
request.headers	Las cabeceras de la solicitud
request.httpVersion	La versión de HTTP de la solicitud
request.method	El método de la solicitud
request.url	La url del recurso o ruta solicitada
response.end([data][, encoding][, callback])	Finaliza la solicitud con una respuesta
response.setHeader(name,value)	Crea una cabecera implícitamente. Si la misma ya existe será reemplazada.
response.statusCode	Controla el código status enviado al cliente si los headers fueron enviados de forma implícita

response.statusMessage	Controla el texto correspondiente al status si los headers fueron enviados de forma implícita
response.write(chunk[,encoding][,callback])	Provee data al cuerpo de una solicitud. Si es llamada sin writeHead, esta se envía junto con las cabeceras implícitas
response.writeHead(statusCode[, statusMessage][, headers])	Envía cabeceras a la respuesta del cliente. Si ya se llamaron los setHeader, estos van a converger con writeHead donde esta tiene la precedencia. Tiene que ser llamada antes de end()

1. *https://nodejs.org/es/*
2. *https://es.m.wikipedia.org/wiki/Reactor_(patrón_de_diseño)*
3. *https://es.wikipedia.org/wiki/ECMAScript*
4. *https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/let*
5. *https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/const*
6. *https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Funciones/parametros_rest*