Entendiendo los conceptos básicos

1. Introducción al módulo

¿Que veremos en este módulo?

- Como funciona la Web y que rol cumple NodeJs
- Creación de un servidor de Node.js
- Módulos que trae el núcleo de NodeJs
- Trabajar con peticiones y respuestas básicas
- Código asíncronico y Event Loop en NodeJs

Official Node.js Docs: https://nodejs.org/en/docs/guides/

Full Node.js Reference (for all core modules): https://nodejs.org/dist/latest/docs/api/

More about the Node.js Event Loop: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/

Blocking and Non-Blocking Code: https://nodejs.org/en/docs/guides/dont-block-the-event-loop/

2. Como funciona la Web

Tenemos usuarios/clientes que interactúan con una página web.

El proceso sería el siguiente:

-Un usuario entra a tu página web, detrás de escena el navegador busca algunos servidores de nombres de dominio. El dominio (nombre del sitio) realmente no es la dirección del sitio, es una codificación en algo que los humanos podamos entender. El servidor en sí tiene una dirección IP que es a lo que detrás de escena se conecta el cliente.

-Una vez encontrado este dominio se envía una petición al servidor con la dirección IP que pertenece a ese dominio.

-El servidor tiene escrito código que nosotros u otras personas hicimos para esa computadora en esa dirección IP y ese código es el encargado de manejar la petición y hacer algo con ella, en general se devuelve una respuesta a esa petición al cliente. La repuesta puede ser código HTML que luego será manejado por el cliente pero puede ser también otro tipo de dato, como JSON, XML, archivos, etc.

-La respuesta no solo tiene contenido, también tiene headers que es meta information adjunta a la respuesta y a la petición, y describe que hay dentro.

-Esa transmisión está hecha bajo un protocolo (que es una forma estandarizada de comunicarse), que son reglas a seguir para lograr la comunicación. Estas reglas son definidas por el protocolo que usemos, HTTP o HTTPS.

-HTTP (Hyper Text Transfer Protocol) es un protoclo para transferir información que es entendida tanto por el navegador como por el servidor. Ahí esta definido como una petición valida debería verse y como la información debería ser transferida del navegador al servidor y viceversa.

-HTTPS (Hyper Text Transfer Protocol Secure) es básicamente lo mismo que HTTP y añade encriptación SSL que logra que la información transmitida este encriptada así nadie colgado en la conexión puede husmear los datos

3. Creando un servidor de NodeJs

https://nodejs.org/api/http.html

https://www.w3schools.com/nodejs/func_http_requestlistener.asp

Normalmente el punto de entrada lo nombramos como server.js o app.js. Lo llamamos así porque es el archivo raíz que hace a nuestra aplicación y es el archivo que va a ejecutar una computadora en la nube en un servidor.

Hay un puñado de funciones y objetos que utilizamos de manera global sin importar nada con JavaScript tanto en el servidor como en el navegador, pero en general, la mayoría de las funcionalidades no están disponibles de forma predeterminada.

Para trabajar con un servidor tenemos que importar algunas funcionalidades del núcleo de NodeJs, otras funcionalidades no vienen integradas y para integrarlas utilizaremos el Node Package Manager (npm).

Algunos de los modelos del núcleo de NodeJs: http, https, fs, path, os.

Los módulos http y https son muy útiles cuando se trata de crear un servidor y trabajar con solicitudes y respuestas http. De hecho, http nos ayuda a iniciar un servidor o también con otras como enviar solicitudes porque una aplicación de node también podría enviar una solicitud a otro servidor, hacer que varios servidores se comuniquen entre sí.

https sería útil cuando queremos lanzar un servidor codificado SSL, donde todos los datos que se transfieren son encriptados. Esta funcionalidad la veremos mas adelante en el curso.

Para importar estas funcionalidades del núcleo de NodeJs asignamos a una variable la importación y utilizamos require("nombre") para importar.

requiere es una palabra reservada, una función especial que NodeJs expone globalmente, por lo que podemos utilizarla por default en cualquier archivo que se ejecute a través de NodeJs

requiere toma un PATH a otro archivo, por lo tanto, podemos importar otros archivos JavaScript o si no tiene una ruta a un archivo, también puede importar un módulo del núcleo. Los PATHS siempre tienen que empezar con ./ para el relativo y / para el absoluto, es importante distinguir que si escribimos un PATH va a buscar un archivo en la carpeta y no un módulo, los módulos globales se importan simplemente escribiendo el nombre. Al igual que React no hace falta agregar la extensión .js al final.

El módulo http tiene una función createServer([options?],requestListener?) que recibe en sus parámetros un request listener. Un request listener es una función que se ejecutará para cada solicitud/petición entrante (Asocia una función a un evento, la request es un objeto IncomingMessage y la repuesta es un objeto ServerResponse). La función createServer devuelve un servidor, así que para poder levantarlo y hacer procesos con el mismo debemos alamcernarlo en una variable y manipularla según nuestra necesidad.

Una función importante de nuestro servidor es la funcion listen(). Esta función inicia un proceso en el que nodejs no saldrá inmediatamente nuestro script, sino que nodejs lo mantendrá en ejecución para "escuchar" las solicitudes entrantes.

listen() argumentos opcionales. El primero es el puerto en el que queremos escuchar. En producción este campo no se utiliza y tomará por default el puerto 80. El segundo es un hostname, por defecto será el de la maquina que lo ejecuta, para nuestra maquina será el localhost.

4. El ciclo de vida de NodeJs y Bucle de eventos

Creamos el archivo y lo ejecutamos con node app.js -> Comienza la ejecución del script (Parsea el código, registra variables y funciones) -> El script queda en loop y nunca termina gracias al concepto de bucle de eventos (event loop, en este caso se ejecuta gracias al método listen() del server)

El Event Loop es básicamente un proceso de bucle que es administrado por nodejs que sigue funcionando mientras haya trabajo que hacer, es decir, mientras haya event listener registrados.

Nuestra aplicación de NodeJs básicamente es administrada por este bucle de eventos, todo nuestro código es administrado por esto y, como se mencionó,

NodeJs usa un enfoque impulsado por eventos para todo tipo de cosas, no solo para administrar el servidor (que es una parte crucial) pero se verá mucho en el curso, por ejemplo, cuando accedamos a una base de datos.

NodeJs utiliza este patrón porque JavaScript se ejecuta en un solo hilo. Entonces, todo el proceso de node usa un hilo en nuestra computadora que está ejecutando el código.

Como podemos imaginar si creamos un servidor de node éste debería de ser capaz de manejar múltiples, miles, decenas de miles o ciento de miles de solicitudes entrantes. Ejecutándose en un solo hilo se detendría y luego haría algo con esa solicitud, lo que lo haría extremadamente lento e ineficiente. Por lo tanto, utiliza este concepto de bucle de eventos donde al final siempre sigue ejecutándose y solo ejecuta código cuando ocurre un determinado evento, así que en general siempre está disponible. Si bien podemos pensar que si llegan dos peticiones al mismo tiempo necesita manejar dos eventos, es muy rápido en el manejo de estas solicitudes y, en realidad, detrás de escena, realiza algunos subprocesos múltiples al aprovechar el sistema operativo.

Resumiendo, el Evento Loop es una parte esencial que básicamente tiene un bucle continuo siempre que haya oyentes (crear un servidor crea un oyente que nunca para). Pero si finalmente se anula el registro (se puede hacer con process.exit(), hace un hard exit) se terminaría. Por lo general el process.exit() no se utiliza porque no queremos cerrar el servidor, si se cierra los clientes no podrán acceder a nuestra página.

5. Entendiendo las peticiones

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

En el callback que le pasamos a nuestro createServer tenemos dos objetos que nos genera NodeJs. Uno es la petición y otro es la respuesta.

El objeto de la petición es el objeto que tiene los datos de la petición cuando visitamos el puerto que indicamos.

El objeto en sí es complejo, tiene muchos datos, funciones que podemos llamar, etc. También podemos observar que viene con headers, los headers son meta-datos/meta-información añadida a la petición (también a respuestas). En ella podemos ver el host, headers que vienen arraigados al navegador, como la respuesta se debe cachear y cosas de ese estilo.

Hay muchos datos que podemos tomar, pero los más importantes son: url (La URL es todo lo que viene después del nombre del dominio/localhost), method, headers.

Una peticion muere cuando se envia una respuesta.

6. Enviando respuestas

Response tiene un montón de métodos por los cuales vamos a poder devolverle al cliente una respuesta a una petición. Una de los métodos es el setHeader("key", "value"), en el cual podemos, por ejemplo, adjuntar el Content-Type que vamos a devolver como metadato.

La función write("CONTENIDO") nos permite escribir datos a la respuesta y funciona escribiendo múltiples líneas de respuesta.

Una vez utilizada estas funciones usamos la funcion end(). No debemos escribir después de cerrar la respuesta ya que genera un error. También podríamos retornar response.end() para terminar en ese momento el bloque de código y asegurarnos que no se ejecutará nada después.

Estas funcionalidades se van a acotar y ser mucho más fáciles ya que utilizaremos el framework Express.Js

7. Headers de peticiones y respuestas

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

8. Ruteo y redirección de peticiones.

https://developer.mozilla.org/es/docs/Web/HTTP/Status

https://www.geeksforgeeks.org/node-js-response-writehead-method/

https://nodejs.org/api/http.html#responsewriteheadstatuscode-statusmessage-headers

https://http.cat/

En vez de escribir información de la petición en la consola, lo que por supuesto no hace mucho, vamos a escribir un servidor web que hace diferentes cosas dependiendo de la ruta que ingresemos. Dependiendo de que venga después del / en la URL vamos a enviar ciertas respuestas al cliente, accedemos con el método url al PATH de la petición para saber dónde ingreso el cliente después del .com, .com.ar, etc (Comienza con / si no hay nada).

El cliente puede, por ejemplo, enviar un formulario con el action="/URL" para luego ser tratado, de esta manera ruteamos un lugar de nuestra página.

Los formularios no solo envían la petición sino que buscan todos los inputs/selects del formulario y lo asocian con un key:value, siendo la key el name que tiene ese campo.

La redirección serviría, por ejemplo, para cuando enviamos un formulario a una URL determinada. Esa URL simplemente maneja la información de la petición y una vez manejada redireccionamos a otra página y desplegar la información que queramos (datos validos, invalidos, volver al inicio y un simple alert box, etc). Hay que tener en cuenta que esa URL que se envía en un formulario para tratar la información debería verificar ciertos parámetros así los usuarios no ingresan simplemente con el /PATH.

El codigo 302 es de redirección. Para redireccionar podemos usar el método writeHead('CODIGO HTTP', {HEADERS});. El método básicamente nos permite escribir meta-information de una sola vez. También los métodos statusCode y setHeader.

Una vez realizado el código cierro la respuesta y la retorno y/o me aseguro que la función termine allí.

Ejemplo: response.writeHead(302, {"Location":"/"});

response.statusCode(302));

response.setHeader({"Location":/"/"})

9. Parsing del cuerpo de la petición

Los datos llegan básicamente como un "stream of data" que tiene un constructor especial en JavaScript.

¿Que es un stream of data y como se conecta con los buffers?

Un stream es básicamente un proceso en curso que es leída por node en trozos y en un momento del tiempo finaliza. Este hecho así ya que en teoría podríamos empezar a trabajar con estos trozos antes de que finalice la lectura de la petición.

Podemos empezar a trabajar con la data enviada desde temprano, sin embargo, con nuestro código no podemos intentar trabajar arbitrariamente con los chunks que llegan. Para organizar estos chunks trabajamos con un buffer (El buffer es como una parada de autobus, donde necesitamos que pare y sube/baje personas y haya un registro/seguimiento de ello). Básicamente es un constructor que permite mantener multiples chunks y trabajar con ellos antes de que termines.

Cuando recibimos un mensaje, antes de retornarlo, darle tratamiento o escribir metadata vamos a escuchar un event listener llamado "on", este evento puede escuchar a varias partes pero a nosotros nos interesa el evento on.("data", "callback").

El evento data será lanzado cada vez que un nuevo chunk está listo para ser leído. El callback es el que va a manejar esa data y recibe en sus parámetros el chunk que llega.

Para leer el body la idea es generar un array en una constante para que no se pueda sobre escribir esa data (si se sobre escribe el contenido del array) y por cada chunk que llega pushearlo en el array.

También tenemos que trabajar con otro evento dentro del on.("end", callback) llamado end. Este evento se lanzará cuando termine de parsear la petición entrante. Según el ejemplo, se lanza el evento cuando todos los chunks ya fueron parseados. También recibe una función que va a manejar esa data que pusheamos en el evento "on".

Para interactuar con todos estos chunks guardados tenemos que utilizar la clase estática Buffer (El buffer, en el ejemplo anterior, sería algo así como una parada del autobus). En el ejemplo creamos una nueva constante y le asignamos el objeto Buffer.concat(body).toString(). Lo convertimos a String como un ejemplo porque SABEMOS que es texto lo que llega, depende del tipo puede variar.

Ejemplo:

```
const body = [];

request.on('data', (chunk) => {

    body.push(chunk); -> <Buffer 6d 65 73 73 61 67 65 3d 64 73 61 73 64>

    console.log(chunk);

});

request.on("end", () => {

    const parsedBody = Buffer.concat(body).toString();

    console.log(parsedBody); -> message=dsasd Dice message porque el input name asociado es ese.

})
```

Todo lo que venimos viendo es raw logic, Express.js nos oculta todo esto, pero para entender cómo y porque usamos Express primero tenemos que saber que está pasando detrás de bambalinas.

10. Entendiendo ejecución de código controlado por eventos

El orden de ejecución del código nos es precisamente en el orden que lo escribimos.

Por ejemplo, los eventos on.(EveList, callback) se pueden seguir ejecutando y nosotros podríamos haber devuelto la respuesta y redirigido al cliente.

Enviar la respuesta no significa que nuestros event listeners estén muertos. Se seguirán ejecutando aún si la respuesta ya fue enviada. También tenemos que tener en cuenta que si la respuesta esta influencia por estos eventos no debemos enviar la respuesta hasta que finalicen. Una solución partiendo del ejemplo sería mover el return de la respuesta adentro del evento on.("end", callback).

Estos son algunos ejemplos de como se ejecutan de manera asincrónica algunas funciones callback de los eventos.

Cuando encuentra un event listener simplemente lo añade internamente y nodeJs se ocupa de ejecutar el calllback cuando se lancen los eventos.

Se podría pensar como que NodeJs tiene registros internos de eventos y los callbacks. Se encarga de manejar toda la lógica de cuando ejecutarlos gracias a sus registros.

Los Eventos se registran internamente y los callbacks asociados se van a llamar cuando se lanzan los eventos. Mientras tanto va a seguir con las siguientes líneas de código. Para evitar esto, si queremos bloquear nuestro stack con un evento tendríamos que asignar la palabra return al evento.

11. Código bloqueante y no bloqueante

writeFileSync se ejecuta, como dice su nombre, de manera sincronía. La sincronía hace que se bloquee el stack y ninguna instrucción más se ejecute hasta que esa línea no haya terminado su proceso.

Si tenemos un texto simple para guardar como hicimos en esta sección lleva un tiempo despreciable, pero si trabajamos con muchísima información podemos llevar a trabajar de manera significativa el stack. Para que esto no suceda tenemos una función que es simplemente writeFile, esta función es asincrónica y recibe además del PATH y el objeto a escribir un callback que a su vez recibe por parámetros un error, el error se ejecuta por si en algún momento fallo asincrónicamente y a su vez ese callback también maneja el objeto recibido.

Ejemplo SIN MANEJO DEL ERROR - Bloqueando el código restante con el evento

```
return request.on("end", () => {
    const parsedBody = Buffer.concat(body).toString();
    const message = parsedBody.split("=")[1];
    fs.writeFile("./message.txt", message, err => {
        response.writeHead(302, {"Location":"/"});
        return response.end();
    })
})
```

12. Node.js - Mirando detrás de escenas

Single Thread, Event Loop & Blocking Code

NodeJs utiliza solo un hilo de ejecución de JavaScript. Un hilo es como un proceso que realiza el Sistema Operativo.

El Event Loop inicia automáticamente por NodeJs cuando nuestro programa se arranca. Es el encargado de manejar los event callbacks, sin embargo, esto no nos ayuda si tenemos que manejar archivos pesados, los event loops ejecutan código de finalización rápida.

FileSystem y otro tipo de operaciones similares se envían a un Worker Pool (Manejado y automatizado por NodeJs), que es el encargado de hacer todo el trabajo pesado. Este Worker Pool esta prácticamente separado de código JavaScript (Ya sea el Event Loop, código, peticiones, etc) y se ejecuta en diferentes hilos e interviene muy de cerca con el SO. Si bien esta separado de JavaScript tiene una conexión con el Event Loop, esto se logra una vez que el Worker finalice y el mismo Worker lanzará el callback apropiado.

Simplificando, a partir de eventos se podría decir que tenemos la siguiente lógica: Se registra el evento al inicio de NodeJs -> FileSystem y módulos de ese tipo se ejecutan en el Worker Pool -> Una vez finalizado ese Worker Pool se ejecuta el callback del evento con el Worker ya retornado.

El Event Loop es al final un loop que es iniciado y manejado por NodeJs. Maneja todos los callbacks y maneja cierto order en el cual tienen que ejecutarse. El orden del Event Loop va de la siguiente manera:

Timers (Puede estar en otra fase y volver a esta sin problemas) -> Callbacks pendientes de tipo Input Output (Disk, Network & Blocking Operations) y Callbacks en cola -> Fase Poll, donde NodeJs busca nuevos eventos de tipo Input Output y ejecuta sus callbacks -> Fase Check, ejecuta setImmediate() callbacks -> Close Event Callbacks. -> Si no hay más manejadores registrados hace un exit.

13. Usando el Node Module System

Para exportar una función utilizamos module.exports = FUNCION/OBJETO y para importarlo utilizamos el requiere y le damos el PATH al archivo donde se encuentra esa exportación.

El archivo exportado es cacheado por Node y no puede ser editado externamente/on the fly.

También podemos exportar más de una función si lo asignamos como un objeto y desestructurando luego en la importación. Ejemplo:

```
module.exports = {
    requestHandler
}
const {requestHandler}= require("PATH");
```

Otra manera puede ser module.exports.NOMBRE = FUNCION/OBJETO

14. Resumen del módulo

¿Como funciona la web?

- Cliente => Peticion => Servidor => Respuesta => Cliente

Ciclo de vida del programa y Event Loop

- Node.js corre de manera no bloqueante JavaScript y usa Event-Driven code ("Event Loop") para ejecutar la lógica
- Un programa de Node deja de ejecutarse tan pronto como no haya más trabajo que hacer.
- Nota: El evento createServer() nunca termina por default.

Código Asincrónico

- JS es no bloqueante
- Usa callbacks y eventos => Los ordenes cambian!

Trabajar con Peticiones y Respuestas

- Parsear data en chunks (Streams y Buffers)
- Evitar "doble respuestas"

Node.js y sus Core Modules

- Node.js viene con multiples core modules (http, fs, path, ...)
- Core modules pueden ser importados a cualquier archivo que los requiera Importación via require("module")

El Node Module System

- Importación vía requiere("./path") o requiere("module") para core modules y core modules de terceros.
- Exportación via module.exports = FUNCION, module.exports.NAME = FUNCION, module.exports = {FUNCIONES}, exports, etc.