Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript (JS ES6), asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables. (Saber más https://es.wikipedia.org/wiki/Node.js)

Veremos por ejemplo que una tarea común para un servicio web de abrir un archivo en el servidor y devolver el contenido al cliente, es manejado por PHP o ASP de la siguiente manera:

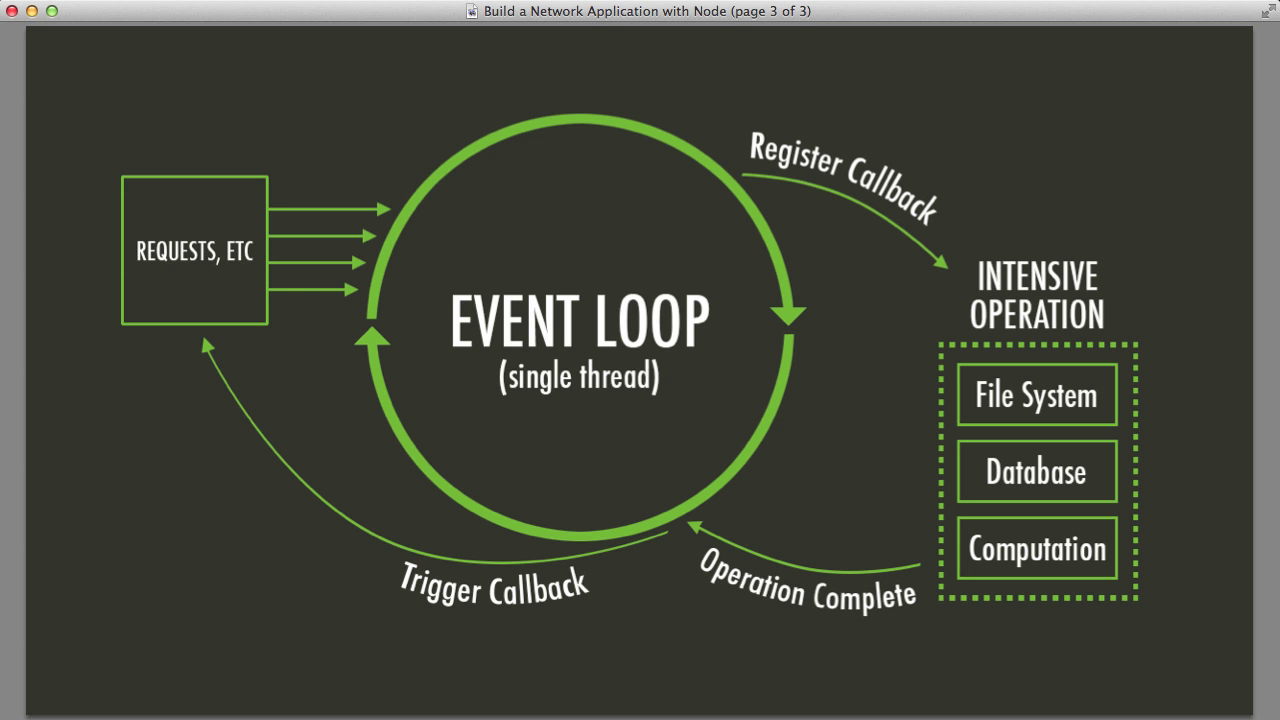
1. Envía la tarea al sistema de archivos de la computadora.
2. Espera mientras el sistema de archivos se abre y lee el archivo.
3. Devuelve el contenido al cliente.
4. Listo para manejar la siguiente solicitud.

Mientras tanto que en Node.js esta solicitud se maneja de la siguiente manera:

1. Envía la tarea al sistema de archivos de la computadora.
2. Listo para manejar la siguiente solicitud.
3. Cuando el sistema de archivos se abre y lee el archivo, el servidor devuelve el contenido al cliente.

De esta forma Node.js elimina la espera y simplemente continúa con la siguiente solicitud.

Node.js se basa en un modelo concurrente no paralelo, ejecutando una programación asíncrona de subproceso único, sin bloque, que es muy eficiente en memoria. Para entender mejor esto, veamos el siguiente diagrama.



Cuando llega un Request al hilo principal de procesamiento del servidor, este es procesado y llama al recurso correspondiente necesario para realizar la operación, que puede ser el sistema de archivos, la base de datos o cualquier otro. Mientras se ejecuta el recurso externo ejecuta la petición el hilo principal de Node.js sigue atendiendo otros request hasta que el callback registra que la solicitud fue cumplida y los recursos solicitados ya están disponibles. De esta forma los datos vuelven al hilo principal y son enviados como respuesta del request inicial.

Para entender esto con un ejemplo, supongamos que un usuario inicia sesión en nuestro servidor en Node.js y para realizar la autenticación llamamos a una base de datos MySQL. Cuando llega el Request de inicio de sesión el código en JavaScript realiza el procesamiento correspondiente y llama a la base de datos MySQL, la cual es un servicio externo a nuestro proceso en Node.js y va a necesitar un tiempo para responder con los datos solicitados. Mientras Node.js espera que la base de datos devuelta la información, este sigue atendiendo otros Request. Cuando la MySQL devuelve lo solicitado, el callback correspondiente de la función que llamo a la base de datos devuelve la información al hilo principal de Node.js y este elabora la respuesta al request del usuario.

¿Porque Node.js?

Entre las ventajas de Node.js podemos destacar:

* Escalabilidad en el procesamiento de datos
* Confiabilidad para operaciones en tiempo real
* Constante mejora tecnológica
* Análisis de errores
* Conexión persistente con el servidor

Y en cuanto a la performance en desarrollo, grandes marcas como Uber, Paypal, Netflix, Trello, Wallmart, Linkedin, eBay, entre otros reportan estos resutlados:

http://inubo.es/noticia/los-10-mejores-ejemplos-de-aplicaciones-node-js-para-empresas

* El código se desarrollo 2x más rápido
* 33% menos de código
* Deploy hasta x5 veces más rápido
* > 2M llamadas RPC por segundo

Empezemos.

El primer paso es instalar Node.js, utilizando las instrucciones de la pagina oficial. <https://nodejs.org/es/>. Tener en cuenta que si usan Debian, Ubuntu, Redhat, CentOS o Fedora los binarios son proporcionados por NodeSource y se tienen que seguir estas instrucciones.

https://github.com/nodesource/distributions/blob/master/README.md

01-HelloWorld

Dento de la primer carpeta tenemos un archivo javascript, en el cual las primeras 2 lineas llamamos a las librerías del sistema operativo y del sistema de archivos, als cuales están incluidas en Node.js. Usaremos la función appendFile para crear un archivo con la información de nuestro CPU. Vemos que en la línea 7 comentada, el callback de la función appendFile esta descripta como function(error) mientras que en la línea enterior el callback esta declarado como una función flecha, sintaxis propia de JS6. Despues de ejecutar esta función se esriben logs con información de neustra PC. Al ejecutar el código con

Node 01-HelloWorld.js

Veremos que el log output de appendFile se ejecuta después de los console.log, a pesar de que este esta antes. Esto se debe a el asincronismo de Node.js. Dado que el sistema de archivos es un recurso externo este tiene un tiempo de respuesta que es manejado por el callback y vemos como el orden en el que se llaman las funciones no es el mismo con el cual se ejecutan.

02-Async-vs-Sync

En este segundo ejemplo vamos a ver como ejecutar en orden el código anterior, utilizando appendFileSync. En este caso las funciones se ejecutan en el orden que están siendo llamadas. Pero el problema es que no tenemos un callback, por lo tanto no tenemos un manejador de erroes para la creación de nuestro archivo. Y el problema principal es que mientras se ejecuta la funcion appendFileSync nuestra aplicación de node.js queda bloqueada hasta que finalice el recurso externo. Por lo tanto las funciones que tienen su análogo sincrono (“Sync”) las debemos reempalzar por alguna otra alternativa, como las promesas que veremos mas adelante, para evitar que nuestro servicio se bloquee.

03-Promises

Las promesas son una forma de manejar los eventos de nuestras aplicaciones en Javascript y por lo tanto en Node.js. Utilizaremos este método para manejar los eventos y callbacks de la ejecución de las funciones que utilizan recursos externos a Node.js.

Este ejemplo utiliza los comandos del sistema operativo para crear un nuevo directorio, escribir un archivo de texto y crear una copia de este. Esto es a modo de ejemplo, si queremos realizar operaciones con archivos y directorios deberíamos usar el FileSystem que provee Node.js. Para continuar haremos uso de la librería child\_process la cual instalaremos con el siguiente comando.

Npm install child\_process

En la primera línea del código llamamos a la funcion exec de la librería externa con un require, y posteriormente una serie de variables que definen los comandos del cmd. En la línea 10 definiremos una funcion, y en la línea 11 retornamos una promesa, con 2 estados, resolve (si esta se resuelve con éxito) y reject (si esta fracasa por causa de un error). Usamos la funcion exec y le pasamos como primer parámetro la iunstruccion que vamos a realizar, el segundo parámetro es un callback. Si hay un problema o un error en ejecutar la instrucción, haremos un reject de la proimesa, si no ejecutaremos resolve para retornar el estado de que la promesa fue resuelta.

Posteriormente ejecutaremos una cadena de promesas en casacada, para realizar todas las operaciones descriptas al comiezo. Oara ejecutar este ejemplo usaremos el comando:

Node 03-Promieses.js

Al ejecutar por primera vez este ejemplo, se creara la nueva carpeta con los 2 archivos correspondientes, y veremos los logs en la consola. Si lo volvemos a ejecutar veremos que tendremos un error ya que la carpeta ya existe por lo tanto tendremos un reject.

03-Hello-Express.js

Express es el framework de node.js mas utilizado, y este nos permite crear aplicaciones web de una forma muy sencilla. Dado que Express es una librería eexterna al paquete de Node.js, debemos instalarla, para ello utilizaremos el gestor de paquetes de Node.js , llamado npm el cual esta incorporado en nuestra instalación de Node.js.

Dentro de la carpeta de esta sección, en la consola del sistema, ejecutaremos el comando

Npm init

Con este inicializamos un nuevo proyecto de node y nos crea un archivo package.json el cual contendrá la descripción de nuestro proyecto y la lsita de la dependencias. Posterieormente ejecutamos

Npm install Express

Con este comando se crea la carpeta node\_modules, la cual contendrá todos los archivos de las librerías y dependencias de nuestro proyecto acual, 03-Hello-Express. La carpeta node\_modules no la debemos incluir en nuestros repositorios git ya que contienen demasiados archivos que no cambian con nuestro proyecto. De un proyecto a otro, podemos utilizar la información del package.json para recuperar todas las dependencias utilizada e instalarlas con un simple comando:

Npm install

Bien, ahora dentro de nuestro archivo 03-Hello-Express.js, crearemos en la línea 2, una nueva aplicación de express, a la cual en la línea 4 le decimos que cuando se realice un get en la ruta ‘/’ de nuestro servidor, se enviara el strong de la línea 6 como respuesta.

En la línea 10 iniliaziamos nuestra aplicación de express, utilizando como primer parámetro el puerto de nuestro servidor, y como callback imprimimos la ruta de nuestra aplicación la cual podremos visualizar en el navegador. Para ejecutar este archivo utilizamos el comando

Node o npm start?

05-Express-Structure

No encontraremos esta carpeta en el repositorio, porque la generearemos con toda la estructura de nuestro proyecto en express utilizando el generador de proyectos de express para ello lo instalamos utilizando el comando

https://expressjs.com/es/starter/generator.html

$ npm install express-generator –g

Teniendo instalado en nuestra PC el generador de express con el siguiente comando generaremos todos los archivos necesarios para un proyecto de Express listo para proudccion.

express --view=ejs 05-Express-Structure

El gestor de plantillas por defecto para Express es jade, pero en nuestro caso usaremos EJS por eso el parámetro view. De este comando “04-Express-Structure” seria el nombre nuestro proyecto en express y el directorio en el que vamos a trabajar.

Para inicializar el proyecto que hemos creado seguimos los comandos:

Cd 05-Express-Structure

Npm install

Npm start

Podremos ver nuestro servicio de node.js y express corriendo en <http://localhost:3000/>. Ahora exploraremos los archivos y directorios creados

* /bin
  + www: Este archivo contiene la configuración inicial de Node.js para nuestra aplicación. Lo más relevante es la deficnion de la variable port que nos dara el puerto en el que va a correr nuestro servidor.
* /public: En esta carpeta se guardan los archivos estáticos del frontend, donde por defecto encontramos 3 subcarpetas : images, javascripts y stylesheets. Mas adelante en el archivo app.js usaremos la funcion express.static para declarar que esta carpeta contiene recursos que pueden ser requeridos por los usuarios por medio de la ruta por ejemplo <http://localhost:3000/stylesheets/style.css>
* /routes : En esta carpeta se encontraran los archivos que manejan las rutas de nuestro servidor, generalmente las peticiones POST y GET. Encontraremos 2, uno que define las rutas para el home y otro el directorio /users.
* /views: Es donde se guardab las plantillas, los archivos EJS que se renderizaran en HTML y se enviaran a los usuarios como respuesta.
* app.js : El archivo principal para iniciar nuestro servidor o aplicación en Node.js. Contiene todos los require principales y la configuración de nuestro servicio.

Se recomienda estudiar el código de este ejemplo para entender lo básico del funcionamiento de las futuras aplicaciones de Node.js y Express.

06-Express-Startkit

En esta sección veremos una estructura de archivos generada de la misma forma que el ejemplo anterior. En este caso iniciaremos el servidor y al hacerlo estaremos haciendo un ‘GET’ en la ruta ‘/’ de nuestro servidor. Este según lo especificado en el archivo /routes/index.js nos devuelve una pagina renderizada en HTML con una variable llamada message. Al poner nuestro nombre en el anvegador, y darle submit al formulario, estamos haciendo un ‘POST’ en la ruta ‘/’ y para este caso en el mismo archivo de rutas tenemos que estamos leyendo el contenido del body del formulario de este request, en este caso la variable req.body.name. Con este parámetro renderizamos un nuevo HTML basándonos en el archivo /views/welcome.ejs.

Vemos entonces en este ejemplo como manejar las distintas rutas, opciones y propiedades de nuestra aplicación al manejar las peticiones HTTP, de las cuales principalmente utilizaremos GET y POST.

*BB-Express-Sockets*

*Ahora vamos a ver un ejemplo de comunicación entre el servidor y*

*Para este ejemplo vamos a eliminar la carpeta public por defecto y vamos a usar bower para instalar la ultima versión de las librerías que usaremos en el frontend directamente desde su reporsitorio oficial con un solo comando.*

*Para ello utilisamos el comando de instalación de bower con npm*

*npm install -g bower*

*teniendo Bower instalado, en la carpeta de nuestro proyecto vamos a instalar las librerías que necesitaremos.*

*Bower install bootstrap  
bower install jquery  
bower install socket.io-client*

*Veremos que se creo una carpeta llamada bower\_components con las librerias instaladas, ahora vamos a decirle a express en el archivo app.js que nuestra carpeta de archivos públicos se llama bower\_components y no public como estaba por defecto.*

*Observaremos una nueva carpeta llamada services en nuestro proyecto, en esta carpeta guardaremos los archivos javascript que definen las funciones que va a usar el backend. En esta carpeta tenemos un archivo llamado delay.js que tiene 3 funciones que solamente generan un log con distintos tiempos de delay. Esto nos servirá para visualizar la comunicación entre el back y el front, mediante las promesas y los web sockets.*

*En el backend necesitaremos instalar la versión de servidor de socket.io, esto ya se instala por defecto al hacer npm install en este proyecto pero debemos incluirla y configurarla en el archivo app.js. Ademas de eso crearemos un servidor http que estará escuchando en nuestro puerto los mensajes recibidos.*

07-Express-MySQL

En este nuevo ejemplo utilizaremos MySQL como base de datos, para ello deberemos tener instalado MySQL y con esto nos dirigiremos al archivo /config/database.js y modificaremos los parámetros con los de nuestro servidor MySQL.

express --view=ejs 07-Express-MySQL

express --view=ejs 10-Express-CRUD